SURVEY OF FLOATING-POINT SOFTWARE ARITHMETICS AND BASIC LIBRARY MATHEMATICAL FUNCTIONS

BY

KENG HO LEE

A THESIS SUBMITTED FOR THE DEGREE OF MASTER OF SCIENCE AT THE UNIVERSITY OF GLASGOW

SEPTEMBER 1973

ACKNOWLEDGEMENTS

I am grateful to Professor D.C. Gilles for his permission to carry out tests on the MULTUM computer and to make use of the facilities in the Computing Department. I am also indebted to Dr J.E. Jeacocke for his guidance and encouragement and for many helpful discussions.

Thanks are also due to Mrs J.K. Clydesdale for her patience with the typing and in general to all the members of the Computing Department of Glasgow University for services rendered, both great and small.

I wish to dedicate this thesis to my wife, grandmother and mother, whose constant encouragement has been of great help to me.

CONTENTS

PART TWO: BASIC LIBRARY MATHEMATICAL FUNCTIONS

PART ONE

INTRODUCTION

In this short span of thirty years man, with the help of digital computers, can perform arithmetic operations at a rate of three million per second. How fast we can really add, multiply, subtract and divide in the future has to do with the performance of tomorrow's computer components, the hardware of the future. Shimuel Winograd (1) in his article has given some insight into the problem involved. In spite of the tremendous increase in the speed of computation, there is little success in overcoming the 'inexact nature' of arithmetic. In the nineteenth century, Mrs La Touche summed up our present problem in her statement 'There is no greater mistake than to call arithmetic an exact science'.

In search of better control over the 'inexact nature', continuing attention has been given to questions concerning the representation of numbers for computers. R.L. Ashenhurst (2) listed the following factors that will affect the choice of a number system.

a) Engineering efficiency.
   This clearly establishes the advantage of the binary system instead of the decimal system.

b) Programming convenience.
   For example, using floating-point rather than fixed-point representation.

c) Detection and correction of machine malfunction.
   For example, introducing extra digits to permit a redundancy check.

d) Assessment of computation error.

The presence of computational error is, of course, inevitable in computation confined to finite resources. Computer designers tend to sacrifice the efficiency in error assessment for the enhancement of performance with

respect to objectives in the first three categories.

It is not the intention of the author to give a detailed study of error assessment. Instead, a survey of the number representations of computers is given. The later part of Chapter One gives an axiomatic approach to floating-point operations. Theorems and definitions quoted are taken from the paper by T.J. Dekker (6). Implementation of both single length and double length floating-point operations is discussed in Chapter Two. Procedures for single length floating-point operations are based on the algorithms given in Kruth (3). Two methods for double length floating-point operations are discussed. Both are based on the theories given in the earlier chapter.

Notations used will be those given in Kruth (3) and those used by Wilkinson (4).

CHAPTER ONE

1.1 NOTATIONS

The positional number system will be referred to throughout the discussion of fixed-point arithmetic, normalised floating-point arithmetic and unnormalised floating-point arithmetic.

DEFINITION (1.1.1)

Positional notation using base b (also called radix b) is defined by the rule

$$(\ldots a_3 a_2 a_1 a_0 . a_{-1} a_{-2} \ldots)_b = \ldots a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0 + a_{-1} b^{-1} + a_2 b^{-2} + \ldots) \quad (1.1.2)$$

The 'dot' between a_0 and a_{-1} is called the radix point. If b=10, we have the decimal system and the 'dot' is commonly known as the decimal point. The a's are called the digits of the representation.

DEFINITION (1.1.3)

The most significant digit is the non-zero digit with the highest subscript.

DEFINITION (1.1.4)

The least significant digit is the digit with the smallest subscript.

1.2 NEGATIVE NUMBERS

There are several different ways of representing negative numbers in a computer.

1.2.1 SIGNED MAGNITUDE REPRESENTATION

This corresponds to the conventional notation by placing a minus sign in front of the number. In the computer whose number representation is in binary, the sign is simply denoted by '0' and '1' for positive and negative numbers respectively. However, we are up against the problem of having two different representations for zero, viz. 'minus zero' and 'plus zero' when they are, in fact, the same value.

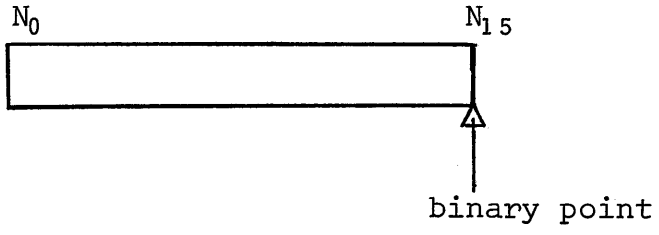
This strongly undermines the practical aspect of computer design which otherwise has a great theoretical advantage over the Complement Notation (3). This inconsistency may be avoided by defining the pattern for 'minus zero' as invalid pattern which may be picked up by the compiler.

1.2.2 COMPLEMENT NOTATION

No explicit sign is attached to the number and calculation is done modulo b^p where p is the number of digits in the computer word and b is the base of the representation. In this case, the question of having two representations for zero is eliminated. However, it should be noted that the complement system is not symmetrical about zero. This not uncommon asymmetric range will, in fact, give rise to overflow in certain types of arithmetic operations. For example, negating the largest negative number gives an overflow as the positive number so formed is not in the range of representation. Shifting right in the complement notation does not necessarily divide the magnitude by the base b. For example, given b=10, (i.e. the decimal system) a number say $(-13)_{10}$ is equal to 99987 if calculation is done modulo 10^5. Shifting right one place gives 99998 which is $(-2)_{10}$. Similarly, for b=2, shifting right does not necessarily divide the magnitude by the base. In fact, the shifting right operation merely produces the functional effect of the function 'entier'.

1.3 FIXED-POINT ARITHMETIC

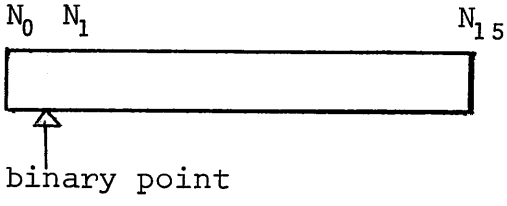
In this mode the computation is so framed that every computed number x lies in a given range depending on the size of the computer word. Take, for example, the 16 bit word MULTUM Computer; fixed-point numbers are represented in 2's complement with binary point after the least significant bit (see figure 1). The computed number x will then lie in the range $[-2^{15}, 2^{15}-1]$.



(Figure 1)

$N_0 \ldots N_{15}$, 16 bit 2's complement with binary point after N_{15}.

All 2^{16} bit patterns are legal, and bit N_0 will always be set for negative values. If we were to interpret with the binary point lying between bit 0 and bit 1, then the number x will lie in the range $[-1, 1)$ (see figure 2).



(Figure 2)

In general, each number will be allowed a fixed number p of binary digits for its representation and we shall say that the computer works with words of p binary digits. If it is necessary to work to a higher precision than one part in 2^p then we may employ numbers which are represented by a multiple of of p binary digits.

1.3.1 ROUNDING ERRORS IN FIXED-POINT COMPUTATION

We now consider the rounding of errors made in the fundamental arithmetic operations. There is no round-off in fixed-point addition and subtraction. However, the computed number x may be outside the permitted range.

In fixed-point multiplication, the product of two p-digit numbers lying in the interval $[-1, 1)$ is, in general, a number requiring 2p digits for its representation. This exact product is rounded off to a p-digit approximation by

adding $\frac{1}{2}2^{-p}$ if number is positive (or by subtracting $\frac{1}{2}2^{-p}$ if number is negative) and discarding digits p+1 to 2p. Let z be the exact product and z' the p-digit approximation of z, then

$$z = z^1+\varepsilon \quad \text{where} \quad |\varepsilon| < \tfrac{1}{2}2^{-p}.$$

Different machines may use different rounding procedures. In some machines, the last p-digits of the exact product are discarded and the p^{th} digit is replaced by 1. This gives an error lying in the range $\pm 2^{-p}$ and therefore a maximum error which is twice that of the earlier method of rounding. It should be noted that the procedure used for rounding will greatly affect the results in floating point computation; especially so if we were to simulate the floating-point operations using the available fixed-point instructions.

The quotient x/y of two p-digit fixed-point binary numbers will lie in the permitted range unless $|y|<|x|$. The quotient will, in general, be a non-terminating number. The exact quotient may be rounded to give a p-digit fixed-point number by the same rounding procedure stated earlier. Note that we need to compute only the first p+1 digits of the exact quotient in order to derive a rounded result.

If z is the exact quotient and z^1 the computed quotient, then $z = z^1 + E$ where $|E| \leq \frac{1}{2}2^{-p}$.

Note: The exact quotient is replaced by the p-digit approximation obtained by adding $\frac{1}{2}2^{-p}$ and retaining only digits 1 to p.

Unfortunately, not all machines produce rounded results. This may lead to serious loss of accuracy in the final result. For example, the MULTUM computer returns a truncated result instead of the rounded one. The error E in this case is less than 2^{-p}.

If x and y are integers, no rounding is performed. Instead, the quotient is the p-digit approximation obtained by truncating digits p+1 to 2p. This is none other than the function 'entier' defined in Algol 60 (12). This

truncating procedure for integer arithmetic is commonly practised in most computers. No special algorithm will then be required for integer arithmetic in the case of a Fortran compiler.

1.4 FLOATING-POINT ARITHMETIC:

In this notation, we let the position of the radix point be dynamically variable ('floating') as the programme is running, and carry with each number an indication of the corresponding radix point position.

The two basic components of a floating-point number, x, are the exponent e(x) and the mantissa (also called fraction part, or significant) m(x). The nomenclature and the precise specifications for e(x) and m(x) vary from one representation to another, but they essentially serve the same purposes. The pair (e(x), m(x)) stands for the number $m(x)b^{e(x)}$ where b is the base (or radix) for the representation. The choice of the base, b, and the number of p-ary digits in m(x) are therefore basic parameters characterising the representation. Hence, given a floating-point number, x, we have

$$x = m(x)\ b^{e(x)} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ 1.4.1$$

Among other characteristics are the range of m(x) and the manner in which m(x) (the mantissa) and e(x) (the exponent) are encoded in the machine, including such things as placement of radix point and representation of negative numbers. For example, the Floating-point Representation proposed by A.A. Garu (5) has m(x) as integers and in the range $-K < m(x) < K$, where K is determined by the number of bits allotted to express m(x). He called m(x) the 'fixed-point part'. Here the implied radix point is after the least significant bit of the p-digit 'fixed-point part'. In fact, Burroughs 5500 (Octal) and Philips Electrologica XB (binary) use this format to represent the floating-point number system (6). Apart from over-flow and underflow, the floating-point systems have this

form in most if not all computers, since the mantissa can always be interpreted as an integer by subtracting a suitable constant from the exponent.

Representation given by equation (1.4.1) is not always unique. In most machines, a certain standardisation is defined in order to make the representation unique. The most common standardisation is the normalisation in which the magnitude of a non-zero mantissa has a lower bound M/b, where b is the base of the representation and M the largest value in the p-digit mantissa. ($M = b^p$ if considered as fixed-point part or $M = 1$ if mantissa is a fraction).

If we take the mantissa m(x) to be a signed fraction, then the process of standardisation consists of selecting as normal that representation in which m(x) has the largest possible absolute value such that

$$|m(x)| < 1.$$

In other words, the radix point appears at the left of the positional representation of m(x). If p is the number of mantissa digits then $m(x)\ b^p$ is an integer and

$$-b^p < b^p m(x) < b^p.$$

This floating-point representation was proposed by Kruth(3) in his 'MIX' computer. The floating-point number x is said to be normalised if the most significant digit of the representation of m(x) is non-zero, so that

$$\frac{1}{b} \leq |m(x)| < 1 \quad \text{where m(x) is the p-digit mantissa}$$

If m(x), the mantissa, is a p-digit signed-2's complement fraction, there is a slight re-adjustment in the range.

For m(x) positive, $\frac{1}{2} \leq m(x) \leq (1-2^{-p})$

and m(x) negative, $-1 \leq m(x) \leq (\frac{1}{2}+2^{-p})$

In general, to tell which of the two normalised floating-point numbers has a greater magnitude, we simply

compare the exponent parts first and then test the fraction parts only if the exponents are equal. Clearly, comparison of this type will be slightly complicated if 2's complement notation is used.

In the later part of the chapter, we will discuss in more detail the various types of floating-point representations.

1.5 NORMALISED CALCULATION

The adjustment to normalised form is justified on the basis that one would like to have numbers uniquely represented and also to preserve as many "significant digits" as possible (since rounding errors are then of relative as well as absolute magnitude b^{-p} to m(x)). The operations of floating-point arithmetic will be studied in detail in this section. The algorithms for floating-point addition, subtraction, multiplication and division will be discussed with the intention of simulating these operations in the I.C.S. MULTUM computer.

1.5.1 NOTATION

The notation found in Kruth (3) will be used. To denote floating-point addition, subtraction, multiplication and division we write $\oplus$, $\ominus$, $\otimes$, $\oslash$ to distinguish the approximate operations from the true ones. The mantissa, m(x) is taken to be a p-digit signed magnitude fraction. Note, however, that in Chapter Two m(x) is taken to be a p-digit (p=24)2's complement fraction since the MULTUM computer uses the 2's complement representation. Minor adjustments are required to implement the algorithms given here. We shall use the notation, (e(x), m(x)) to denote the floating-point x such that

$$(e(x), m(x)) = m(x)\ b^{e(x)-q}$$ where q is the excess (c.f. equation 1.4.1)

For example, in the MULTUM computer with excess q equal

to 128, the number $(128, 0.10000_2)$ denotes the floating-point number 0.5,

$$\text{i.e. } (128, 0.10000_2) = \tfrac{1}{2}\, 2^{128-128} = (0.5)_{10}.$$

Similarly, the number

$$(127, 0.10000_2) = \tfrac{1}{2}\, 2^{127-128} = (0.25)_{10}$$

This reduces e(x), the exponent to a positive integer. The assignment of sign to the exponent is avoided and arithmetic operations (addition or subtraction) on the exponent are greatly simplified by introducing the excess q.

ADDITION/SUBTRACTION

Let x and y be two normalised floating-point numbers, such that

$$x = m(x)\, b^{e(x)-q},$$

and $$y = m(y)\, b^{e(y)-q}, \text{ where } \tfrac{1}{2} \leq m(x),\ m(y) < 1.$$

Assume $|x| \geq |y|$, for floating-point addition we represent the sum z, where $z = m(z)\, b^{e(z)-q}$ such that

$$m(z)\, b^{e(z)-q} = m(x)\, b^{e(x)-q} \oplus m(y)\, b^{e(y)-q}$$

also written as

$$(e(z), m(z)) = (e(x), m(x)) \oplus (e(y), m(y))$$

Assuming $|x| \geq |y|$, we need to divide m(y) by the amount $b^{e(x)-e(y)}$ so as to align the radix point for a meaningful addition (or subtraction). This is equivalent to a shift right operation of e(x)-e(y) positions. Hence we have

$$m(z) = m(x) \oplus m(y)/b^{e(x)-e(y)}.$$

The mantissa, m(z) is then normalised.

1.5.2 ALGORITHM FOR ADDITION/SUBTRACTION

The same algorithm may be used for floating-point subtraction if -y is substituted for y. The base b is

assumed to be even. Note, however, that negating a negative number, y, may give an overflow in subtraction.

Step 1. The floating-point numbers are first unpacked to give the exponents and fractional parts (mantissae).

Step 2. Test if $e(x)<e(y)$, interchange x and y if true.

Step 3. Set $e(z) = e(x)$.

Step 4. Obtain $e(x)-e(y)$ and test if $e(x)-e(y) \geq p+2$.

Step 5. m(y) is shifted right $e(x)-e(y)$ places.

Step 6. Round m(y) to p+2 digits to minimise the length of register which is needed for addition/subtraction in Step 7.

Step 7. Addition/subtraction.
Set $m(z) = m(x) \pm m(y)/b^{e(x)-e(y)}$

Step 8. Normalise m(z) and round z into the final answer.

1.5.3 REMARKS

(a) In Step 4, we can jump out of the routine instead of going to Step 7 as the operands are assumed to be normalised. Also note that testing $e(x)-e(y) \geq p$ will be sufficient if truncation is used instead of rounding in Step 6 (and also in the normalisation procedure).

(b) In Step 5, scaling right involves the shifting of m(y) up to p+1 places. This implies that an accumulator capable of holding 2p+1 base b digits to the right of the radix point is required. Normally m(y) is truncated to p+2 digits to minimise the length of register which is needed for addition/subtraction in Step 6. The procedure for truncating to p+2 digits is as follows: If m(x) and m(y) have the same sign, replace m(y) by sign m(y) $b^{-p-2}\lfloor b^{p+2}|m(y)|\rfloor$; if

m(x) and m(y) have opposite signs, replace m(y) by sign m(y) $b^{-p-2}\lceil b^{p+2}|m(y)|\rceil$

where $\lfloor x \rfloor = \max k$, $k \leq x$, k being an integer

and $\lceil x \rceil = \min k$, $k \geq x$, k being an integer (1.5.4)

The transformation has no effect if m(y) = 0 and e(x)-e(y) < 3 since $b^{p+2}m(y)$ would then be an integer. Assuming m(y) ≠ 0, and e(x) ≥ e(y)+3 and since x is normalised, we have x ≠ 0, clearly

$$|\, m(x) + m(y) \,| > \frac{1}{b} - \frac{1}{b^3} > \frac{1}{b^2}$$

This implies that the leading non-zero digit of m(x) + m(y) must not be more than two positions to the right of the radix point and the digit which governs rounding must not be more than p+2 positions to the right of the radix point. This is equivalent to zeroing out the digits of m(x) + m(y) which are more than p+2 digits to the right of the radix point.

(c) If x and y are not normalised, deleting Step 4 will give the required result. This is very impractical since it will require a very large accumulator (with about as many digits as the range of exponents). Alternatively, we can normalise x and y between Step 1 and Step 2. Another solution is to change Step 4 to Step 4'. If m(x) = 0, set e(z) = e(y) and m(z) = m(y) and jump to Step 7; otherwise, if e(x)-e(y)≥2p+1, set m(z) = m(x) and go to Step 7. This implies a large accumulator of 3p digits is needed unless some pre-normalisation has been done.

1.5.5 ALGORITHM FOR NORMALISATION

Assume b is even and |f| < b. This algorithm converts an unnormalised floating-point number to the

normalised form. The fraction, m(x) is rounded to p-digits before packing the exponent e(x) and the fraction part, m(x) to give the desired representation. The following rounding procedure is used:

$$\text{If } m(x) > 0 \quad \text{set } m(x) = b^{-p}\left\lfloor b^{p}f + \tfrac{1}{2}\right\rfloor$$

$$m(x) < 0 \quad \text{set } m(x) = b^{-p}\left\lceil b^{p}f - \tfrac{1}{2}\right\rceil$$

(refer equation 1.5.4.)

If 2's complement notation is used, the rounding procedure will be slightly different.

Alternatively, we can truncate the fraction part (i.e. mantissa) m(x) to p-digits after normalising m(x). However, the result obtained will have a larger probable error.

Step 1. Test if $|m(x)| \geq 1$,
Overflow if true, jump to Step 4.
If m(x) = 0 set e(z) = 0 and go to Step 7.

Step 2. Test if m(x) normalised,
i.e. test if $|m(x)| \geq \frac{1}{b}$, go to Step 5 if true.

Step 3. m(x) not normalised, scale left.
m(x) is shifted one place to the left and decrease exponent by 1, go to Step 2.

Step 4. Overflow occurs, m(x) is shifted one place to the right, and increase exponent by 1.

Step 5. Round (or truncate) m(x) to p-digits.
In rounding to p-digits, overflow may occur when $|m(x)| = 1$, go to Step 4.

Step 6. Test for exponent overflow and underflow.
If underflow set m(z) = e(z) = 0.
If overflow set largest representable floating-point number and jump to error condition.

Step 7. The fractional part m(z) and the exponent e(z) are put together into the desired output representation.

1.5.6 FLOATING POINT MULTIPLICATION AND DIVISION

Assume x and y to be normalised floating-point numbers such that

$$x = m(x)\ b^{e(x)-q}$$

and

$$y = m(y)\ b^{e(y)-q}$$

where $\frac{1}{b} \leq |m(x)|, |m(y)| < 1$

and b,the base of the representation,is even.
Let z be the product of x and y.
It follows that either m(z) = 0 or

$$\frac{1}{b^2} \leq |m(z)| < 1 .$$

In division, m(x) is shifted right one place before division. This ensures that $|m(y)| \geq |m(x)|$. The division m(y) is first tested for zero divide. Error condition is returned if zero divide.

Algorithm (for floating-point multiplication and division)

Step 1. The floating-point numbers, x and y, are first unpacked.

Step 2. Test if m(y) = 0.
If true jump out of routine and set error condition for division.

Step 3. (i) For Multiplication:

Set e(z) = e(x) + e(y)-q where q is the excess; then multiply the fraction parts.

$$m(z) = m(x) \times m(y).$$

(ii) For Division:

Set e(z) = e(x) - e(y)+q+1, where q is the excess.
Shift m(x) one place to the right and divide by m(y),
i.e. m(z) = m(x)/m(y)*b

Step 4. Jump to normalisation routine.

Remarks: Rounding overflow cannot occur after division. However, rounding overflow is possible in multiplication.

For example: Let m(x) = (255, 0.10001)

m(y) = (255, 0.11110),

then x * y = (255, 0.1111111110),

rounding gives (256, 0.10000).

If $0 \leq e \leq 255$, overflow occurs.

1.6 FLOATING-POINT ARITHMETICS FOR ALGORITHMIC LANGUAGES

Two types of numbers, real and integers used in Algorithmic languages (e.g. ALGOL) are implemented on the computer by means of floating-point numbers and fixed-point numbers, respectively. However, the set of fixed-point numbers is not a proper subset of the set of floating-point numbers, while in the mathematical sense the set of integers is a proper subset of real numbers. Further, two types of arithmetic operations (floating-point operations and fixed-point operations) are needed to describe the one type of mathematical operation. This divergence between the mathematical(and ALGOL) number concept and hardware usage imposes on ALGOL translation the necessity of handling types (real or integer) dynamically (5). Garu (5) described a floating-point representation and normalisation scheme which avoids, for the most part, the need for dynamic type handling.

The floating-point number representation defined is the same as that described in section (1.4) except that m(x), the 'fixed-point' part (to denote the difference from 'mantissa', which is a fraction) is an integer in the range

$-K < m(x) < K$, where K is an integer determined by the number of bits used to represent m(x).

This range has the advantage that a number with an integral value N in this range then has a floating-point representation (N, 0). However, normalisation may change the

representation to one in which there is no simple relation between the floating-point representation and the fixed-point representation. The following normalisation algorithm was suggested to avoid this (5).

Let (m(x), e(x)) be any floating-point number, r the number of leading zeros in the fixed-point part, and s the number of trailing zeros, then:

1. If e(x) = 0, the number (m(x), 0) is normal.
2. If e(x) > 0, shift off a number of leading zeros equal to the smaller of b and r and compensate by decreasing the exponent b by this number.
3. If e(x) < 0, shift off a number of trailing zeros equal to the smaller of -e(x) and s and compensate by increasing the exponent b by this number.

The normal form is that representation of a number in which e(x) is as close as possible to 0. Table 1 illustrates the patterns of the representation. For simplicity, the decimal system is taken.

Clearly, integers can be recognised immediately by the property $e(x) \geq 0$ and if the integer is in the range $-K<N<K$, by $e(x) = 0$. Fixed-point numbers can be extracted from the normalised floating-point number with an integral value. The problem of not having the set of fixed-point numbers as a proper subset of floating-point numbers is resolved. If the set of fixed-point numbers is restricted to those that can be represented in the fixed-point part range, it is no longer necessary to provide for two sets of machine operations to correspond to the set of mathematical ones. Normalised floating-point addition, for example, will also serve as fixed-point addition and may, therefore, be referred to simply as addition. The use of this kind of fixed-floating arithmetic will considerably simplify the handling of types in ALGOL and other algorithmic languages. In particular, the problem of the dynamic handling of types for the most part is satis-

TABLE 1 Examples Illustrating the Algorithm

	INITIAL FORM				NORMAL FORM		
ACTUAL NUMBER	m(x)	r	s	e(x)	m'(x)	e'(x)	INTEGER
12345678.	12345678	0	0	0	No change		Yes
1234567800.	12345678	0	0	2			Yes
1234.5678	12345678	0	0	-4			No
12340000.	12340000	0	4	0			Yes
1234000000.	12340000	0	4	2			Yes
123400.	12340000	0	4	-2	00123400	0	Yes
12.34	12340000	0	0	-6	00001234	-2	No
5678000000	00005678	4	0	6	56780000	2	Yes
56.78	00005678	4	0	-2	00005678	-2	No

factorily solved.

1.7 UNNORMALISED FLOATING POINT NUMBERS

A.L. Asherhurst and N. Metropolis (7) suggested an arithmetic in which some of the difficulties of conventional floating point arithmetic are avoided by not normalising the numbers used except where absolutely necessary. Algorithms for floating-point computer arithmetic are described in which fractional parts are not subjected to the normalisation conventions. These algorithms give results in a form which furnishes some indication of their degree of precision. The unnormalised arithmetic system must be flexible enough to permit adjustment to be determined by a combination of automatically applied rules and programme-determined options. The MANIAC III Computer in the University of Chicago was designed to achieve this in hardware (2).Based on a single unnormalised exponent-coefficient number format, several varieties of arithmetic manipulation are made possible through the inclusion of operations which employ 'specific point', 'normalised' and significance adjustment rules for results. The unnormalised format permits smooth transition to zero since there exists a multiplicity of "relative zero" with coefficient 0 and arbitrary exponent. There is also an incidental advantage in the avoidance of 'exponent underflow' by never allowing a result to be adjusted so that its exponents exceed the lower limit.

In Metropolis (7), (8) type of arithmetic the probable error in the absolute value of a number is implied by the presence of leading zero digits rather than being stated explicitly. Any number so represented is implied to have an error of plus or minus half the least significant digit.

In addition and subtraction, the exponents of the two operands are made equal by shifting the number having the smaller exponent and then adding (or subtracting) the

fractions. Thus the sum will have no fewer leading zero digits than is justified by the accuracy of the operands. The result is left unnormalised so that all the digits which appear in the register are significant. Thus at any stage of the computation, a meaningful result is produced with no ambiguity as to the significance of the digits.

In multiplication, the factor having the larger fraction is first normalised. The other operand enters the multiplication with its leading zero digits. The product so formed will have about as many leading zero digits as the least accurate factor.

In division the divisor is first tested for zero divide. The correct number of leading zero digits in the quotient is produced by first shifting the dividend fraction right until it is less in absolute value than the divisor fraction. The quotient fraction is formed by dividing this shifted dividend fraction by the normalised divisor fraction. The exponent is incremented each time a right shift is performed and exponent overflow and underflow are tested before assembling the quotient fraction and exponent.

This sort of unnormalised floating-point representation gives an extra dimension to the process, which can be exploited for purposes of significance monitoring.

W.G. Wadey (8) gave a comparative study of these types of arithmetic as compared to the conventional floating-point arithmetic. In addition to the Metropolis floating-point arithmetic given above, floating-point arithmetics with probable error computation are discussed (refer (8)). In the later type, the probable errors in arithmetic computations are computed as the root mean square of the probable errors of the two operands, and are explicitly carried in the last significant digit. This sort of arithmetic can be used as a check on the

accuracy of computations with other arithmetics. It can also be used to perform at once a computation where error analysis is difficult or long in order to determine how many significant digits are obtainable. The validity of this arithmetic is based on the assumption that the errors of the operands are not correlated and the error distributions are normal distributions (5).

1.8 'EXACT ARITHMETICS'

Notations used in the sections that follow are taken from Wilkinson (4). The expression fl(A) is used to correspond to an expression involving floating-point numbers and the arithmetic operations, +, -, ×, /. (Denote +, -, x or / by *.) The definitions and theorems given are taken from Dekker's paper (6).

DEFINITION (DEKKER) 1.8.1

The floating-point number system R is defined as

$$R = \{x | x = m(x)\, b^{e(x)},\ |m(x)| < M,\ -D < e(x) < E\} \quad (1.8.2)$$

where M is a positive integer,
D and E are positive integers or infinite when we disregard overflow and underflow,
b is the base of the system R,
m(x) is the mantissa and e(x) the exponent.

Note: The m(x) and e(x) are integers with a certain range and M depends on the number of mantissa digits.

DEFINITION (DEKKER) 1.8.3

The floating-point operation corresponding to * is 'faithful' if, for all x and y, the result fl(x * y) equals either the largest element of R smaller than or equal to x * y, or the smallest element of R larger than or equal to x * y.

i.e. $z = fl(x * y)$

where $z = \text{Max } z^1$ or $z = \text{Min } z^1$

$z^1 \leq x * y$ $\qquad$ $z^1 > x * y$

When x * y lies between two successive elements of R, either one will do. When x * y R, then z=fl(x*y)=x*y

i.e. result exact. When x*y is outside the range of R, then z = fl(x*y) is the largest or the smallest element of R.

DEFINITION (DEKKER) 1.8.4.

The floating-point operation corresponding to * is optimal (or properly rounding) if, for all x and y, the result fl(x*y) is an element of R nearest to x*y.

Hence, if z = fl(x*y), and if * is optimal,
z is uniquely defined except when x*y lies halfway between two successive elements of R;
in which case, an optimal operation may round up or down.

DEFINITION (DEKKER) 1.8.5.

Floating-point addition is 'properly truncating' if it is commutative (i.e. fl(x+y) = fl(y+x)) and, for all x and y satisfying $|x| \geq |y|$, the result fl(x+y) equals the largest element R smaller than or equal to x+y if $y \geq 0$ or the smallest element of R larger than or equal to x+y if $y < 0$.

The definition uniquely determines the result. When x+y is not an element of R, the truncation is in the direction of -y.

DEFINITION (DEKKER) 1.8.6.

Floating-point subtraction is 'properly truncating' if, for all x and y, we have fl(x-y) = fl(x+y'), where y' = -y and the floating-point addition is properly truncated.

DEFINITION (DEKKER) 1.8.7

Floating-point addition and subtraction are 'super faithful' if,for each x and y, the result fl(x±y) is obtained by properly rounding or by properly truncating.

Remarks: 1. Floating-point numbers considered are not normalised. To obtain a faithful addition

and subtraction, the result must only be normalised before it is truncated or rounded, c.f. Kahan (12). Optimal addition and subtraction can be perfectly well formulated using an accumulator having no more than two guarding digits (Kruth (3)).

2. Floating-point arithmetic is not associative i.e. $(x+y) + z \neq x + (y+z)$ (refer Knuth (3), page 198).

NOTATION

For any real r, round (r) denotes an integer closest to r. Let $z = fl(x+y)$ and $z \in R$.

If floating-point operation * is optimal (properly rounding) then z can be represented by

$$z = m(z)\, b^{e(z)}$$

where $m(z) = \text{round}\,(x*y\, b^{e(z)})$,

provided that no overflow or underflow occurs.

Let us now consider some of the basic rules which are valid for normalised floating-point operations as described in the previous section. Note that the floating-point number system R defined by equation (8.2) still holds except when underflow occurs in normalisation.

$$\left.\begin{array}{ll} \text{(i)} & \text{addition is commutative,} \\ \text{i.e.} & fl(x+y) = fl(y+x) \\ \text{(ii)} & fl(x-y) = fl(x+(-y)) \\ \text{(iii)} & fl(x+y) = 0 \quad \text{if } y = -x \\ \text{(iv)} & fl(x+0) = x \\ \text{(v)} & fl(x-y) = -f(y-x) \end{array}\right\} \rightarrow \quad (1.8.8)$$

We shall prove rule (v) of equation(1.8.8).

Proof:

$$\begin{aligned} fl(x-y) &= fl(x+(-y)) && \text{from rule (ii)} \\ fl(x+(-y)) &= fl((-y)+x) && \text{from (i) commutative} \\ &= -fl(y+(-x)) && \text{from (iii)} \\ fl(x-y) &= -fl(y-x) && \text{from(ii)} \end{aligned}$$

Note: The laws would not be strictly true if two's complement notation were used for the fraction-parts in floating-binary arithmetic instead of signed-magnitude representation.

Consider the following round procedure.

Define

$$\text{round}(x, p) = x \text{ rounded to p-digits}$$

$$= b^{e(x)-p}\left\lfloor b^{p-e(x)}x + \tfrac{1}{2}\right\rfloor \quad \text{if } b^{e(x)-1} \leq x < b^{e(x)}$$

$$= 0 \quad \text{if } x = 0$$

$$= b^{e(x)-p}\left\lceil b^{p-e(x)}x - \tfrac{1}{2}\right\rceil \quad \text{if } b^{e(x)-1} \leq -x < b^{e(x)}$$

Clearly, a floating-point operation corresponding to * is optimal if this rounding procedure were used to obtain the result fl(x*y), provided no exponent overflow or underflow occurs.

The following relationships also hold if the above rounding rule is used.

$$\begin{aligned} fl(x+y) &= \text{round}(x+y, p) \\ fl(x-y) &= \text{round}(x-y, p) \\ fl(x\times y) &= \text{round}(x\times y, p) \\ fl(x/y) &= \text{round}(x/y, p) \end{aligned} \qquad (1.8.9)$$

provided no overflow occurs.

From equations (1.8.8) and (1.8.9) the following identities are true:

$$fl(x\times y) = fl(y\times x)$$

$$fl((-x)\times y) = -fl(x\times y)$$

$$fl(1\times y) = y$$

$$fl(x\times y) = 0 \quad \text{if } x = 0 \text{ or } y = 0$$

$$fl(-x/y) = fl(x/-y) = -fl(x/y)$$

$$fl(0/y) = 0$$

$$fl(x/1) = x$$

$$fl(x/x) = 1$$

The aim in floating-point routines is to preserve as many of the ordinary mathematical laws as possible. As shown above, if the operations are defined according to a consistent set of conventions, many of the mathematical laws will hold true in spite of the inexactness of floating-point operations.

1.9 THEOREMS ON EXACT ARITHMETIC

To estimate the errors involved in floating-point operations the exact arithmetic of the floating-point numbers is considered. In addition to the system R, defined by equation (1.8.2), we assume the floating-point numbers are normalised. Four exact operations will be considered, namely, addition, subtraction, multiplication and division.

1.9.1 EXACT ADDITION

Let x and y be given elements of R (x, y normalised) and let

$$z = fl(x+y).$$

We can find a correction term, zz, satisfying the exact relation

$$z + zz = x+y$$

THEOREM 1.9.2 (DEKKER)

If the floating-point number system R has the form

$$R = \{x | x = m(x)\, b^{e(x)},\ |m(x)| < M,\ -D < e(x) < E\}$$

and if (i) b = 2 or 3

(ii) M is a multiple of b

(iii) Floating-point addition optimal and subtraction faithful

(iv) $e(x) \geq e(y)$ where x, y are elements in R

then the correction term zz given by the equation

$$\begin{aligned} &a)\ w = fl(z-x) \\ &b)\ zz = fl(y-w) \end{aligned} \qquad (1.9.3)$$

such that $x+y = z+zz$ where $z = fl(x+y)$.

Proof: Given $x, y \in R$ and $e(x) \geq e(y)$
we can write

$$x = m(x)\, b^{e(x)}$$

$$y = m(y)\, b^{e(y)}.$$

To prove the theorem, all we need to show is that fl(y-w) and fl(z-x) are exact.

Since subtraction is faithful, it remains to show that

(i) $z-x \in R$

(ii) $y-w \in R$

Proof (i) We need to consider 2 cases,
a) $e(z) = e(x)+1$
b) $e(z) \leq e(x)$

Case (ia) $e(z) = e(x)+1$

Let $z = m(z)b^{e(z)}$ where $m(z) \doteq$ round $(x+y)\, b^{-e(x)}$

i.e. $m(z) =$ round $(m(x)\, b^{e(x)} + m(y)b^{e(y)}\, b^{-(e(x)+1)}$

Let $d = e(x)-e(y)$
then

$$m(z) = \text{round}(m(x)/b + m(y)/b^{d+1}) \quad \ldots\ldots\ldots\ldots (*)$$

Now $z-x = m(z)\, b^{e(x)+1} - m(x)\, b^{e(x)}$

$$= \left(m(z)\, b - m(x)\right) b^{e(x)}$$

then

$$z-x = \mu b^{e(x)} \text{ where } \mu = bm(z)-m(x).$$

Consider

$$\mu = bm(z)-m(x)$$

$$= bm(z)-m(x) - \frac{m(y)}{b^d} + m(y)/b^d$$

$$|\mu| \leq |bm(z)-m(x)-m(y)/b^d| + |m(y)/b^d|$$

$$\leq b|m(z)-m(x)/d - m(y)/b^{d+1}| + m(y)/b^d$$

$$\leq b|m(z)-m(x)/d - m(y)/b^{d+1}| + m(y)/b^d$$

i.e. $|\mu| \leq b/2 + M$ from equation (*)

but

$b \leq 3$ and μ is integral.

We have

$$|\mu| \leq M.$$

Also M is a multiple of b. Hence

$$|\mu| < M.$$

By definition of R, therefore, (z-x) is an element of R.

Case (iia). To prove $y-w \in R$ where $w = fl(z-x)$

Now

$$y-w = m(y)\, b^{e(y)} - m(w)\, b^{e(x)}, \text{ where } m(w) = \mu \text{ and } e(x) = e(y)+d,\ d \geq 0$$

$$= m(y) - b^{d}\ b^{e(y)}$$

$$\therefore\ y-w = \text{integer} \times b^{e(y)}.$$

If $|y-w| > |y|$; then x would be closer to x+y than z, contradicting the assumption that floating-point addition is optimal. Therefore

$$|y-w| \leq |y|$$

which implies that $|\gamma| < M$, where $\gamma = m(y) - \mu b^{d}$.

Hence $y \in R$.

Case (ib): $e(z) \leq e(x)$.

Overflow may occur when $e(z) = e(x)$
i.e. $z = fl(x+y)$ takes the largest value in the range of R nearest to x+y (refer definition (1.8.4)).

Similarly, we can prove (i) $z-x \in R$ and $y-w \in R$.
This completes the proof of theorem.

1. If R is the system of normalised floating-point numbers, Theorem (1.9.2) still holds provided no underflow occurs in forming zz.

2. If M is not a multiple of b, Theorem (1.9.2) breaks down. For example, if b = 3, $M = \frac{1}{2}(3^{t}+1)$ (balanced ternary system, (3)) and x = y = M-1
then $z = fl(x+y) = 3^{t}$
and $z-x = 3^{t} - \frac{1}{2}(3^{t}-1) = \frac{1}{2}(3^{t}+1) = M$
which implies $z-x \notin$ (not an elemental of) of R.

3. If base b > 3, Theorem (1.9.2) fails too.
For example: b = 10, M = 100, and x = y = 99,

then $z = fl(x+y)$

$= 20 \times 10^1$

$= 200$

and $z-x = 200-99 = 101$ which is not in R

i.e. $z-x \notin R$.

4. Theorem does not hold if addition is only faithful.

For example, let b=2, M=16, x=15, y=13/B2 and z=16. (i.e. we take $z = \min_{k \geq x+y} k$),

refer definition of faithful (1.8.3),

then $w = z-x = 1 \in R$

but $y-w = \frac{13}{32} - 1 = -19/32 \notin R$.

Theorem (1.9.2) can be extended to any b and M, provided that the mantissa range is enlarged to accommodate the intermediate value w. As w may be anonymous, (for example in the ALGOL 60 statements

z:= x+y, zz:=y-(z-x), w remains anonymous) and some systems have an enlarged mantissa range for the anonymous floating-point values, an extended theorem has practical application.

If floating-point addition, which is optimal, is replaced by properly truncating addition (see Definition (1.8.5)and subtraction is faithful, Theorem (1.9.2) holds without any restriction on b and M and without requiring an enlarged mantissa range for w. We have the following theorem

THEOREM (1.9.4) DEKKER

If $R = \{x | x = m(x)\, b^{e(x)},\ |m(x)| < M,\ -D < e(x) < E\}$

(i) floating-point is properly truncated and subtraction faithful;

(ii) $x, y \in R$ and $e(x) \geq e(y)$;

(iii) $z = fl(x+y)$;

then the correction term zz defined by

$$z + zz = x + y$$

is given by the equations

(i) w = fl(z-x)
(ii)zz = fl(y-w)

Proof of Theorem (1.9.4) is similar to the proof given for Theorem (1.9.2). Since subtraction is faithful, we need only to show that

(i) z-x ε R
(ii) y-w ε R

Remarks:

1. Consider example given in section 1.9.3, remark 4. b=2, M=16, x=15, y=13/32, then z=15 and not 16 if addition is properly truncating and zz = 13/32-D = 13/32 ε R.

2. From theorems (1.9.3) and (1.9.4) the following corollary holds.

COROLLARY (1.9.5)

If $R = \{x|x = m(x)\ b^{e(x)},\ |m(x)| < M,\ -D < m(x) < E\}$ where b=2 or 3 and M is a multiple of b, and if addition is 'super faithful' and subtraction faithful, then the correction term zz is given by the equations

(i) w = fl(z-x)
(ii)zz = fl(y-w)

such that z + zz = x + y.

Proof for Corollary (1.9.5)

By definition of 'super faithful' (see definition (1.8.7) and from theorems (1.9.2) and (1.9.4) the result follows.

If w = fl(x-z) and zz = fl(w+y)..... (1.9.6) the theorems (1.9.2) and (1.9.3) remain valid if the floating-point number system R is symmetric. This definitely will hold true if the signed-magnitude representation was used to represent the mantissa, (as x ε R implies -x ε R). Writing w = fl(x-z) and zz = fl(w+y) instead of (1.9.3) has some practical advantage. For

example, in Algol 60 we write

$$z := x+y,$$
$$zz := x-z+y.$$

Again w remains anonymous. For applications this formula is preferred as many compilers produce a slightly faster code for (1.9.6) than (1.9.3).

THEOREM (1.9.7) (MOLLER-KNUTH)

If R has the form $R = \{x | x = m(x)\, b^{e(x)},\ |m(x)| < M,\ -D < e(x) < D\}$ and floating-point addition and subtraction are optimal then for all x and y in R and for z = fl(x+y) the correction term zz is given the equations

$$w = fl(z-x), \quad z1 = fl(y-w)$$
$$v = fl(z-w), \quad z2 = fl(v-x)$$
$$zz = fl(z1-z2)$$

provided overflow and underflow does not occur.
We can write this as

$$x+y = z+fl(z1-z2)$$

i.e.
$$x+y = fl(x+y) + fl(z1-z2).$$

In Knuth (3), the theorem is proved for normalised floating-point numbers provided that no overflow or underflow occurs. Instead of calculating zz, -z2 is computed, and added to z1, i.e. we have

$$x+y = fl(x+y) + fl(z1+z2).$$

For a proof of this theorem, refer Knuth (3).

Alternatively, the same lines of the proof of the theorem can be given. Since subtraction is optimal, we need only to prove that (i) $y-w \in R$, (ii) $v-x \in R$ and (iii) $z1-z2 \in R$.

(1.9.8) EXACT MULTIPLICATION

Let $R = \{x | x = m(x)2^{e(x)},\ |m(x)| < 2^t$ .....(1.9.9)

For exact multiplication we need to find zz such that

$$z + zz = x + y.$$

NOTATION:

R(t) is used to denote a binary floating point t-digit number system.

THEOREM (DEKKER) 1.9.10

If $R = \{x | x = m(z)2^{e(x)}, |m(x)| < 2^t\}$,

floating-point addition and subtraction are optimal and multiplication is faithful, then for all x and y in R split into head and tail according to

$$h(x) = \text{round } (m(x)2^{-t_1})\, 2^{e(x)+t_1}$$

and $$t(x) = x - h(x)$$

where t_1 and t_2 are given by

$$t_2 = \text{entier } (t/2) \text{ and } t_1 = t - t_2$$

the formulae

$$p = fl(h(x) \times h(y))$$

$$q = fl(h(x) \times t(y) + t(x) \times h(y)) \quad (1.9.11)$$

$$r = fl\ t(x) \times t(y)$$

$$z = fl(p+q)$$

$$zl = fl((p-z) + q) \quad (1.9.12)$$

and $$zz = fl(zl+r) \quad (1.9.13)$$

yield z and zz satisfying

$$z + zz = x \times y.$$

Proof: Given x and y are split such that

$$h(x) = \text{round } (m(x)2^{-2t_1})\, 2^{e(x)+t_1}$$

where h(x) is the 'head' of x and is an element of $R(t_2)$ very near x (if $x = m(x)2^{e(x)}$, m(x) is normalised) since h(x) is obtained by rounding,

t(x) is an element of $R(t_1 - t)$ as $t_1 = t - t_2$.

Then we have

$$h(x) \times h(y) \in R(2t_2)$$

$$h(x) \times t(y),\ t(x) \times h(y) \in R(t-1) \qquad (1.9.14)$$

$$t(x) \times t(y) \in R(2t_1 - 2)$$

Since t_2 = entier $(t/2)$

and $t_1 = t - t_2$

then $2t_2 \leq t$ and $2t_1 - 2 \leq t-1$.

Since $h(x) \times t(y)$ and $t(x) \times h(y)$ are representable as elements of $R(t-1)$ with the same exponent, their sum is an element of $R(t)$ and formulae (1.9.11) are exact as floating point operations are faithful.

From (1.9.12), since $|p| \geq |q|$ we have $e(p) \geq e(q)$ and by Theorem (1.9.2)

$$z + zl = p+q$$

as floating-point addition and subtraction are optimal. From these assumptions, it also follows that zl is representable as an element of $R(t-1)$ with the same exponent as r. Hence

$$zl + r \in R$$

so that zz defined by

$$z + zz = x \times y$$

is obtained from

$$zz = fl(zl+r).$$

Hence proved.

Also zz is (almost) negligible within machine precision with respect to $x \times y$. This implies that for $t \geq 2$

$$|zz| \leq |x \times y| \alpha 2^{-t}/(1-\alpha 2^{-t}) \qquad (1.9.15)$$

where $\alpha = 2$ if t is even and $\alpha = 3$ otherwise.

Proof:

If $x = 0$, or $y = 0$ then

$zz = z = 0$

then (1.9.15) holds.

Assume $x \neq 0$ and $y \neq 0$

$h(x)$ and $h(y) \neq 0$ and $p+q \neq 0$.

Let $\varepsilon = t(x)/x$, $\eta = t(y)/y$, $\delta = zl/(p+q)$.

The exact product is given by

$$z + zz = x \times y$$

where $z = p+q - zl.$

Now $zz = x \times y - z$

$= xy - p+q + (p+q)\delta$

$= xy - (1-\delta)\ (p+q)$

$= xy - (1-\delta)\ \{h(x)h(y)+h(x)t(y)+h(y)t(x)\}$

$= xy - (1-\delta)\{h(x)\ (h(y)+t(y))+h(y)t(x)\}$

$= xy - (1-\delta)\ \{h(x)y+h(y)t(x)\}$

also $h(x) = x-t(x)$

$\therefore\ zz = xy -(1-\delta)\{xy - t(x)y+h(y)t(x)\}$

$= xy -(1-\delta)\{xy-t(x)(y-h(y))\}$

$= xy -(1-\delta)\{xy-t(x)t(y)\}$

Substituting for t(x) and -(y),
we get

$$zz = xy -(1-\delta)\ (xy-\varepsilon\eta\times y)$$

$$= xy\ \{1-(1-\delta)(1-\varepsilon\eta)\}$$

$$zz = (x\times y)\{(1-\varepsilon\eta)\delta+\varepsilon\eta\}$$

Since floating-point addition is optimal, we have

$$|\delta| \leq 2^{-t}/(1+2^{-t})$$

Similarly, $|\varepsilon|\ ,\ |\eta| \leq 2^{-t_2}/1+2^{-t_2}$

since $h(x) = \text{round}\ \{m(x)\ 2^{-t_1}\}2^{e(x)+t_1}$

and $t(x) = t-h(x).$

Hence

$$|zz| \leq |x\times y|\{2^{-t}/(1-2^{-t})\}\ \{1-2^{t-2t_2}(1-2^{1-t})/(1-2^{-t_2})^2\}$$

and $t_2 = \text{entier}\ (t/2)$

$\therefore\ 2^{t-2t_2} = \alpha-1$

and for $t \geq 2$

$$1 + 2^{-t_2} \geq 1 + \alpha 2^{-t} \geq 1 + 2^{1-t}$$

we get

$$|zz| \leq |x\times y|\{2^{-t}/(1-2^{-t})\} \quad \{1+ \frac{\alpha-1}{1+\alpha 2}-t\}$$

Hence

$$|zz| \leq |x\times y|\alpha 2^{-t}/(1+\alpha 2^{-t}).$$

1.10 CONCLUSION

The pair of formulae (1.9.6) and z = fl(x+y) is the basic algorithm for exact addition of two floating-point numbers. In the next chapter, this algorithm is used to obtain the algorithm for double-length addition. The role of the terms x and y is often interchanged when $|x|<|y|$ so that the condition (iv) in Theorem (1.9.2) is satisfied. It should be noted that in the actual implementation on the MULTUM computer, the conditions in Theorem (1.9.4) are assumed. This is due to the fact that floating-point numbers are truncated after normalisation in the MULTUM computer. Further, it is assumed that no overflow and underflow occurs in our calculation.

For exact multiplication, the formulae (1.9.11) to (1.9.13) are used to obtain the algorithm for double length multiplication. The extra condition (1.9.15) is assumed. However, the bound for zz may be modified to give a smaller bound by the following algorithm.

After calculating z and zz as (1.9.15),an exact addition of z and zz is performed.

Let u = z and uu = zz,
then calculate

$$\begin{aligned} z &= fl(u+zz) \\ \text{and} \quad zz &= fl((u-z)+uu) \end{aligned} \qquad (1.10.1)$$

Then z and zz still satisfy

$$z + zz = x \times y$$

and since addition is optimal, we now have

$$|zz| \leq x \times y |2^{-t}/(1+2^{-t}).$$

The bound for zz is only 2 or 3 times smaller than (1.9.15). It may not be worthwhile in terms of three extra additions or subtractions to form z and zz.

CHAPTER 2

2.0 INTRODUCTION

The implementation of both single precision and double precision floating-point arithmetics is based on the discussions found in the first chapter. One major difference between this and the theories described in Chapter 1 is that the two's complement representation is used instead of the sign-magnitude representation. Only minor changes are required to implement the floating-point arithmetics in the MULTUM ALP 2/3 computer. For example, in truncation, the difference in the direction of truncating must be appropriately accounted for and care be taken if negating a negative number.

A brief description of the MULTUM ALP 2/3 processor and its arithmetic instructions will be given in the next two sections. Literature regarding the specifications of ALP 2/3 processor and the description of the symbolic language used can be found in references (9) and (10).

Two methods of making double-precision arithmetic available are discussed in the later part of the chapter. The methods studied are Dekker's method (6) and the one using the conventional method (3). The second method needs special fixed-point arithmetic, namely 'Long Multiply' and 'Long Divide' in addition to those fixed-point instructions available.

2.1 GENERAL DESCRIPTION (of ALP 2/3)

The main arithmetic, decision making, and logical operations of the computer system are performed by the Arithmetic and Logical Processor (ALP 2/3). The processor works on 16-bit words and up to 256K words of immediate access memory can be addressed. Arithmetic operations are performed using 2's complement arithmetic.

There are seven 16-bit programme accessible registers, namely, registers A, B, X, Y, S, P and the condition register (CR). Register A is the main programme accessible register of the ALP and acts as the accumulator in single-length arithmetic operations. Register B is used for double-length operations and it forms the least significant half of the double length register E (AB). Information can be fetched and stored from memory using registers A, B or E. Registers X and Y are general purpose registers and are used for address calculation and for inter-register arithmetic and logical operations. The register S is the sequence control pointer and holds the address of the next instruction in the programme sequence. The Data pointer register P is used as the base address for the current set of memory held registers. (See page 13, Ref. (9)).

2.2 FIXED POINT FORMATS AND INSTRUCTIONS

2.2.1 SINGLE LENGTH INTEGERS

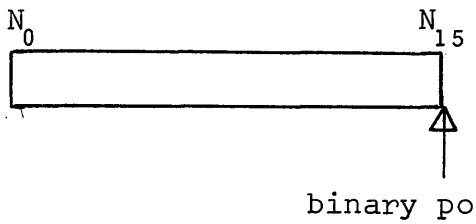


figure 1

Single precision fixed point values are held as 16 bit 2's complement binary numbers with the assumed binary point immediately to the right of the least significant bit. This gives a range of integral value of

$$-2^{15} \leq i \leq 2^{15}-1.$$

The range is assymetric about zero and there exists an integer -2^{15} which has no representable negative.

DOUBLE PRECISION FIXED-POINT FORMAT

The double precision fixed point values are held as 32 bit 2's complement binary numbers with assumed binary point immediately to the right of the least significant bit of the least significant word. The range of values representable by this format is $\left[-2^{31}, 2^{31}-1\right]$. Similarly, -2^{31} will have no representable negative.

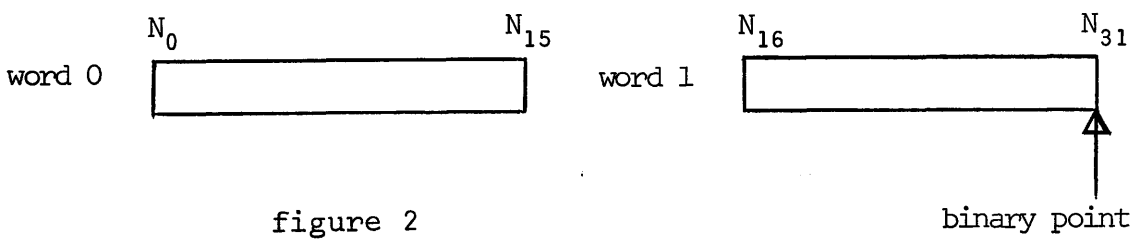


figure 2

Note that the double precision format is slightly modified

in division.

2.2.2 FIXED-POINT INSTRUCTIONS

In fixed-point operations, 'overflow' is said to occur when the result of an operation is outside the representable range. In single precision, fixed-point addition and subtraction (ADDA, SUBA) overflow occurs when the result r is less than -2^{15} or greater than or equal to 2^{15}.

The instruction MLTA multiplies two single-length precision fixed-point values and produces the correct double precision fixed-point result in the range

$$-(2^{30}-2^{15}) \leq i \leq 2^{30}$$

It should be noted that the double precision fixed-point result obtained by the operation 'MLTA' is a 30 bit product occupying bit position 2 to 31 (inclusive) of the 32 bit register E (AB). The sign bit is duplicated in bit 0 and 1. Interpreting the result as a fixed-point fraction (i.e. radix point between bit 0 and bit 1), the product formed must be shifted left one place to give the correct fraction. Bit 0 ≠ Bit 1 if we multiply (-1) by (-1).

The only serious problem arises with division. The instruction DIVE treats the dividend as a special format and divides it by a single precision fixed-point divisor, and produces a single-length fixed-point quotient and a single-length fixed-point remainder. There are two interpretations of the double precision fixed-point format in division.

a) Fraction.

All operands and results have an assumed binary point

immediately to the right of the most significant bit

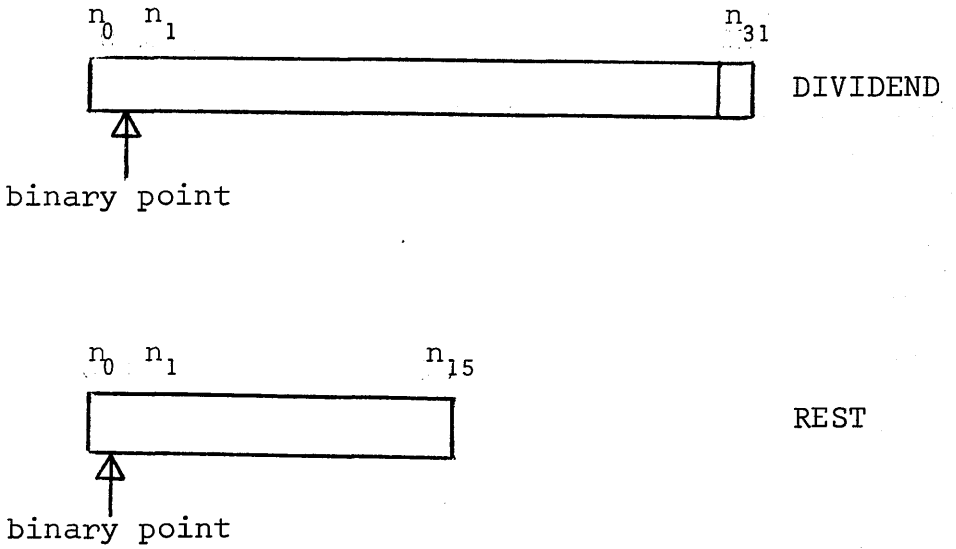


Figure 3

Interpreted in this sense, the dividend is taken to be a fraction occupying 31 bits (0-30). Overflow occurs if the divisor is less than or equal to the dividend.

b) Integer.

The dividend has an assumed binary point immediately to the right of bit 30. The remainder and quotient are single length integers with binary point immediately after bit 15.

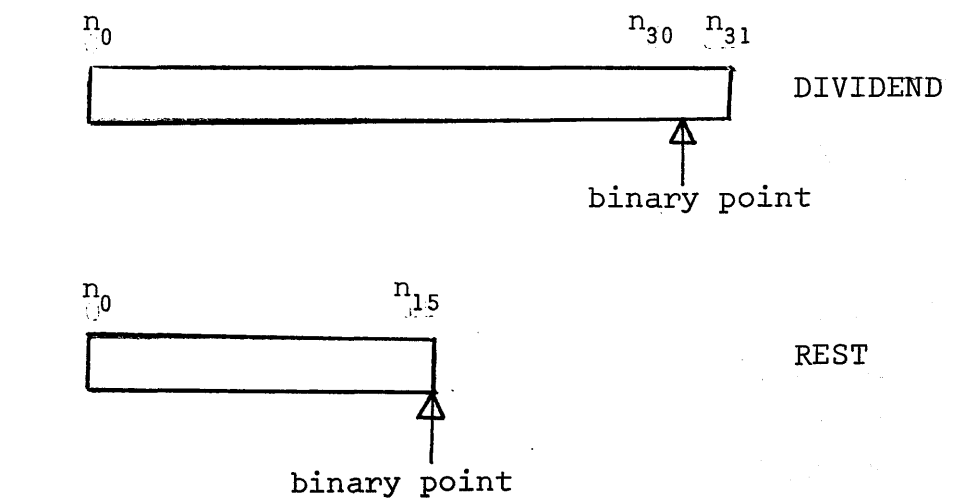


Figure 4

Given any double-length integer, the dividend must be shifted left one place before division is carried out.

Two errors may occur with DIVE,

(i) Zero divide,

(ii) Overflow.

An overflow situation occurs when the ratio of the absolute values of the dividend and divisor is greater or equal to 2^{15}.

Remarks:

There is a lack of uniformity in fixed-point formats for double-length operations. This can be resolved most simply by using the format for 'MLTA' as a standard. In this, bit 0 is equal to bit 1 for all valid patterns. Arithmetic overflow should be set if any fixed-point operation produces a double-length result with bit 0 not equal to bit 1. However, the instruction 'DIVE' is defined to work on bit 0 to 30, so either re-define DIVE to operate on bit 1 to bit 31 or always precede 'DIVE' by a logical shift left of 1 bit.

2.3 FLOATING-POINT FORMATS AND FLOATING-POINT INSTRUCTIONS

2.3.1 FLOATING-POINT FORMAT (SINGLE PRECISION)

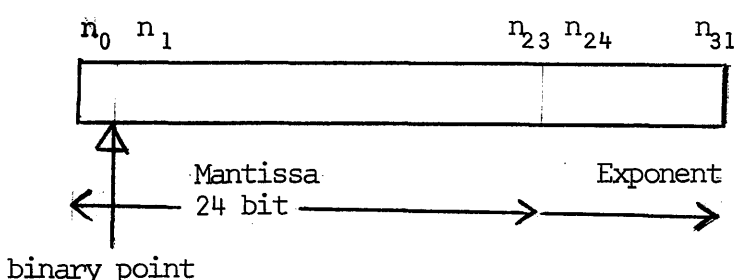


Figure 5

$n_0 \ldots n_{23}$ 24-bit mantissa m(x), 2's complement with assumed binary point between n_0 , n_1 . The 24 bit signed 2's complement fraction will be in the following range

m(x) positive $\quad \frac{1}{2} \leq m(x) \leq (1-2^{-23})$

m(x) negative $\quad -1 \leq m(x) \leq -(\frac{1}{2}+2^{-23})$

$n_{24} \ldots n_{31}$ 8-bit exponent stored biased 128, base 2. Range of exponent e(x) is in the range $-128 \leq e(x) \leq 127$ and stored as 0, 255

The above range for mantissa assumes standardisation to either all 32 bits zero, representing value zero, or bit 0 not equal to bit 1 ($n_0 \neq n_1$). All other bit patterns are said to be non-standard but all have the value 2^{exponent} x mantissa assigned. The bit pattern 80000000_{16} for single precision floating-point number is undefined.

2.3.2 DOUBLE PRECISION FLOATING-POINT FORMAT

Double precision floating-point arithmetics are realised by software (using the available fixed-point instructions or using the available single precision floating-point instructions). This is dependent on the format used. The following double precision floating-point format was first proposed by I.C.S.

2.3.3 DOUBLE PRECISION FORMAT (by I.C.S.)

Double precision floating-point numbers will occupy four words of store and are allocated as detailed in figure (6).

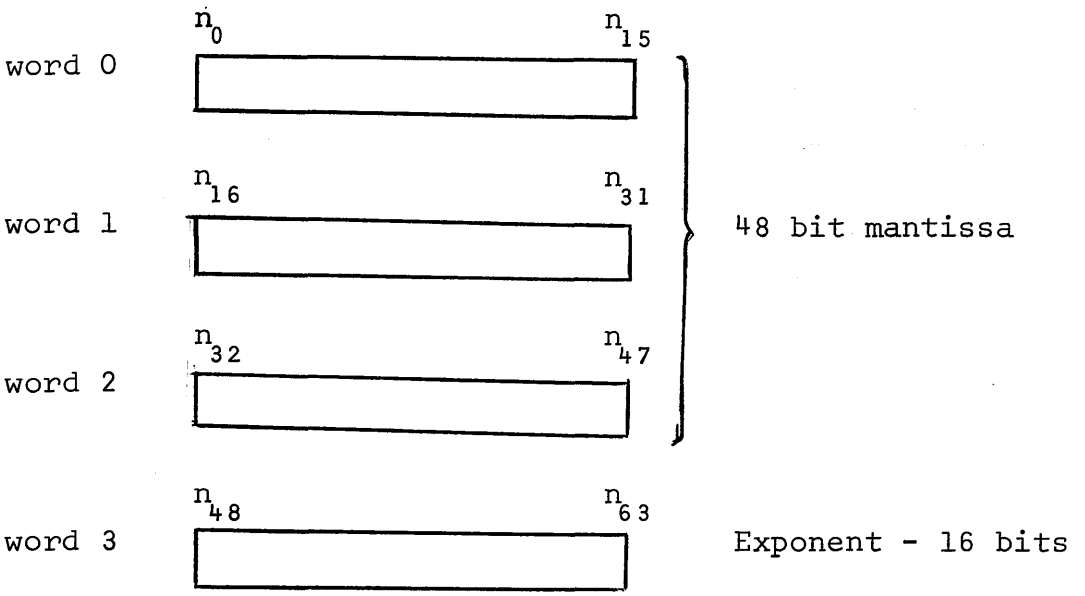


Figure 6

The mantissa is a 48 bit signed 2's complement fraction and is in the following range:

$$m(x) \text{ positive} \quad \tfrac{1}{2} \leq m(x) \leq (1-2^{-47})$$

$$m(x) \text{ negative} \quad -1 \leq m(x) \leq -(\tfrac{1}{2}+2^{-47})$$

The exponent is held as a 16 bit number in excess 32768. Zero is held as four zero words. For standardised floating-point numbers, bit 0 and bit 1 are not equal. All other bit patterns are non-standard. The bit pattern $8000\ 0000\ 0000\ 0000_{16}$ for double-precision floating-point number is undefined.

2.3.4 DOUBLE PRECISION FORMAT WITH REDUCED EXPONENT

Alternatively we could have the following set-up as a double precision floating-point format (refer to figure 7).

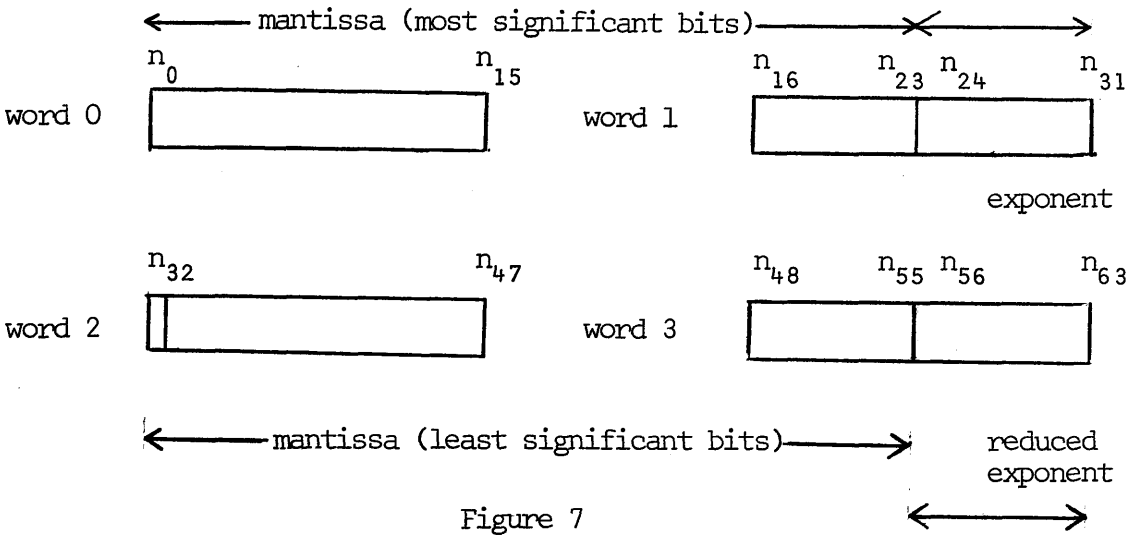


Figure 7

$n_0, \ldots n_{23}, n_{33}, \ldots n_{35}$ 47 bit mantissa 2's complement with assumed binary point between n_0 and n_1. Range of mantissa

$$\left[-1, -(\tfrac{1}{2}+2^{-46})\right], 0, \left[\tfrac{1}{2}, 1-2^{-46}\right]$$

$n_{24} \ldots n_{31}$ 8 bit exponent, stored excess 128, base 2. Exponent range is $[-128, 127]$ and stored as $[0, 255]$

n_{32} always set to 0 and is not part of mantissa.

$n_{56} \ldots n_{63}$ 8 bit reduced exponent, stored excess 128, base 2. The reduced exponent takes the value (exponent -23)if this is greater than, or equal to, zero and $n_{33} \ldots n_{55}$ are not all zero. If reduced exponent is zero then all of n_{32} to n_{63} are cleared

The above range for the mantissa assumes standardisation to either all 64 bits equal to zero, representing value 0

or bit 0 not equal to bit 1 (i.e. $n_0 \neq n_1$). All other bit patterns are non-standard, but still have the value 2^{exponent} x mantissa assigned. The pattern $8000\ 0000\ 0000\ 0000_{16}$ is the 'undefined value' for double precision floating-point number.

Remarks:

1. The proposed new format having the advantage;
   a) Conversion from single precision to double precision is done by appending 2 cleared words.
   b) Software implementation will be satisfactory. As seen in Chapter 1, a straightforward coding of Dekker's method (6) can be used. In the later part of the chapter we shall discuss the methods involved.
   c) Word 2 and word 3 form a (possibly non-standard) single-precision floating-point number.
   d) Implementation of multiple-precision floating-point software is simplified.
2. The loss in the range of double precision is scarcely a disadvantage since
   a) there was a lack of compatibility between single and double precision floating-point formats. Conversion from one to the other is hazardous.
   b) the hidden danger of catastrophic loss of precision when using values of the size of 10^{1000} far outweighs the small advantage.

2.3.5 FLOATING-POINT INSTRUCTIONS

Floating-point arithmetic for single precision is realised by hardware in the ALP 3 processor. The floating-point instructions for addition, subtraction, multiplication, standardisation, etc., are simulated here for the ALP 2 processor and also to test the accuracy of the floating-point arithmetics performed by the hardware (in the ALP 3 processor). The following mnemonics FADDF, FSUBF, FMLTF, FDIVF, FSTND, FFIXF, FFIXI, FFLTF, FFLFI and FNEGF are used to denote floating-point add, subtract, multiply, divide, standardise, fixed fraction, fixed integer, float fraction, float integer and floating-point negate respectively.

2.3.6 ADDITION/SUBTRACTION (FADDF/FSUBF)

The flowchart for addition/subtraction is given in figure (8). Coding of the algorithm is straightforward and is given in the appendix (1). The routine expects the operands to be in standardised format.

The alignment of the binary points before adding the mantissa is done by an arithmetic shift (of the mantissa with smaller exponent) by, at most, 23 places. If more than 23 right shifts are required the smaller operand is set to zero. This procedure ensures that addition or subtraction is properly truncating (refer Chapter 1, definitions (1.8.5) and (1.8.6)). For example, consider the extended mantissa m(x) and m(y).

Let $m(x) = (45678900)_{16}$

$m(y) = (FFFFFE00)_{16}$

If we allowed a right shift of more than 23 places (bits)

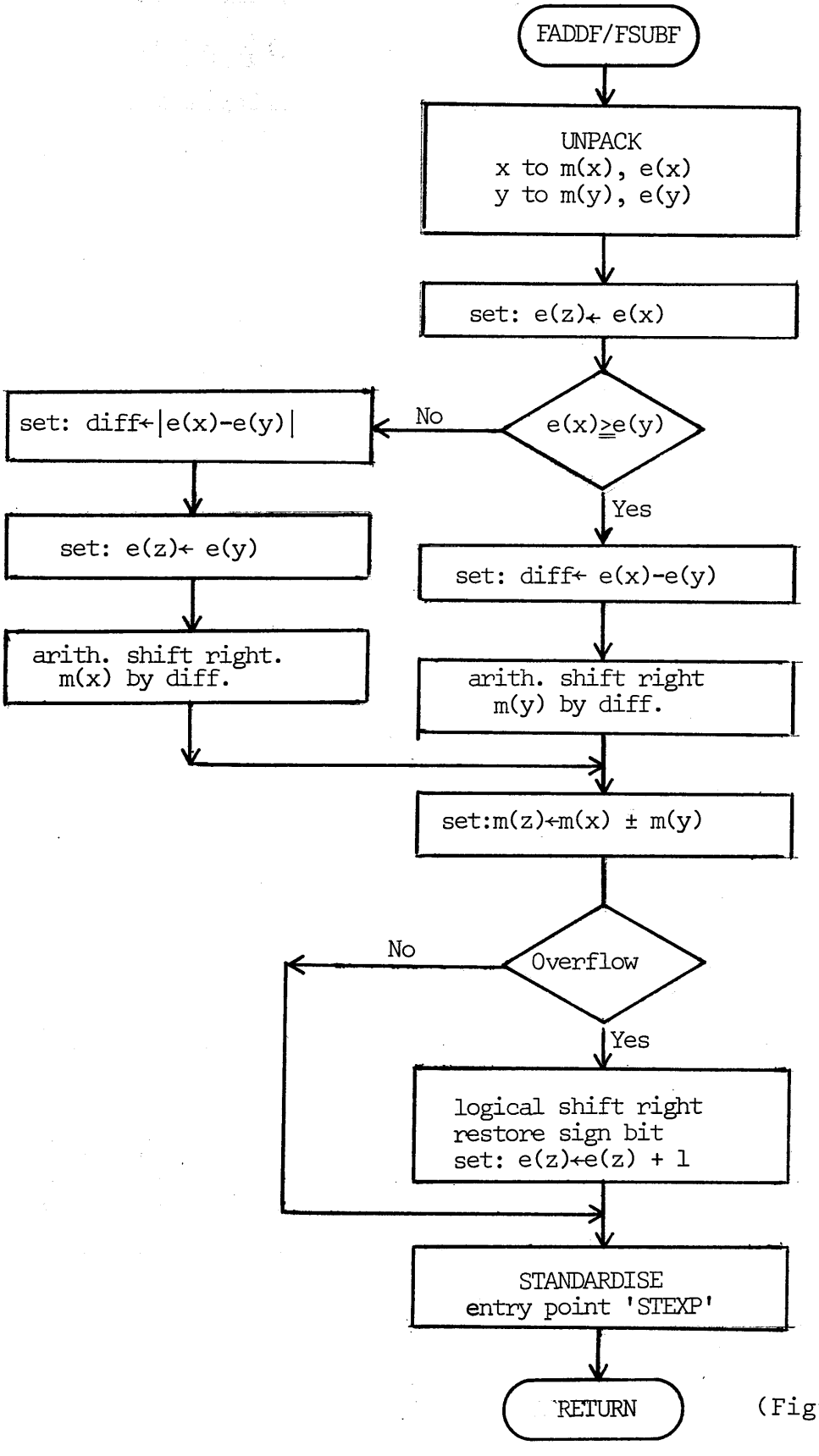
say $e(x) = 81_{16}$ and $e(y) = 69_{16}$ (i.e. an exponent difference of 24) then the sum $m(x) + m(y)$ is equal to $(45678FF)_{16}$. After standardisation and truncation, the floating-point result is equal to $(45678781)_{16}$. If a right shift of less than or equal to 23 places (bits) is allowed, the sum so formed would then be $(45678981)_{16}$. In this case, we return the value of the larger operand if the difference in exponent is greater than 23. Clearly we can see that the operation in the latter case is properly truncating (as given by definition (1.8.5) and (1.8.6) since

$$(45678981)_{16} > (45678781)_{16}.$$

2.3.7 MULTIPLY (FMLTF)

The flowchart for the multiplication routine is given in figure (9). It is a straightforward adaptation of the algorithm given in Chapter 1. The main difficulty in the procedure is in Step 3 when we multiply the mantissae. The multiply instruction available is MLTA. However, this instruction multiplies two single-length numbers (16-bits) to form a 32-bit product (refer section 2.2.2). Interpreting the mantissa and product as fixed-point fractions, the correct fraction (product) is obtained by a left shift of one bit, (provided no overflow occurs). The extended mantissae are first truncated to two 16-bit words before multiplication. The truncated mantissae are then multiplied to give a 32-bit product. A significance of at most 16-bits is attained by this process. To get a full 24-bit significant (digit) fraction, the extended mantissae are split into two

FLOWCHART: ADDITION/SUBTRACTION
(FADDF/FSUBF)



(Figure 2)

16-bit words and the following procedure is followed.

Let m(x), m(y) be the two 32-bit mantissae (extended, last 8 bits all zero). Then m(x) is split into $m_1(x)$ and $m_2(x)$, where $m_1(x)$ is the most significant half and $m_2(x)$ the least significant half. Similarly, m(y) is split. Then,

$$m(z) = m(x)\, m(y)$$

$$= \{m_1(x) + \varepsilon m_2(y)\} \quad \{m_1(y) + \varepsilon m_2(y)\}$$

$$= m_1(x)m_2(y) + \varepsilon\{m_1(x)m_2(y) + m_2(x)m_1(y)\} + \varepsilon^2 m_2(x)m_2(y)$$

where ε is the reciprocal of the word-size.

The value $\varepsilon^2 m_2(x)m_2(y)$ is too small to be significant in the final result since ε^2 is a right shift of 30 bits (places).

Hence we have

$$m(z) \simeq m_1(x)m_1(y) + \varepsilon\{m_1(x)m_2(y) + m_2(x)m_1(y)\}$$

Remark:

Adjustment of $m_1(x)$ is required if m(x) is negative. This is best explained by an example.

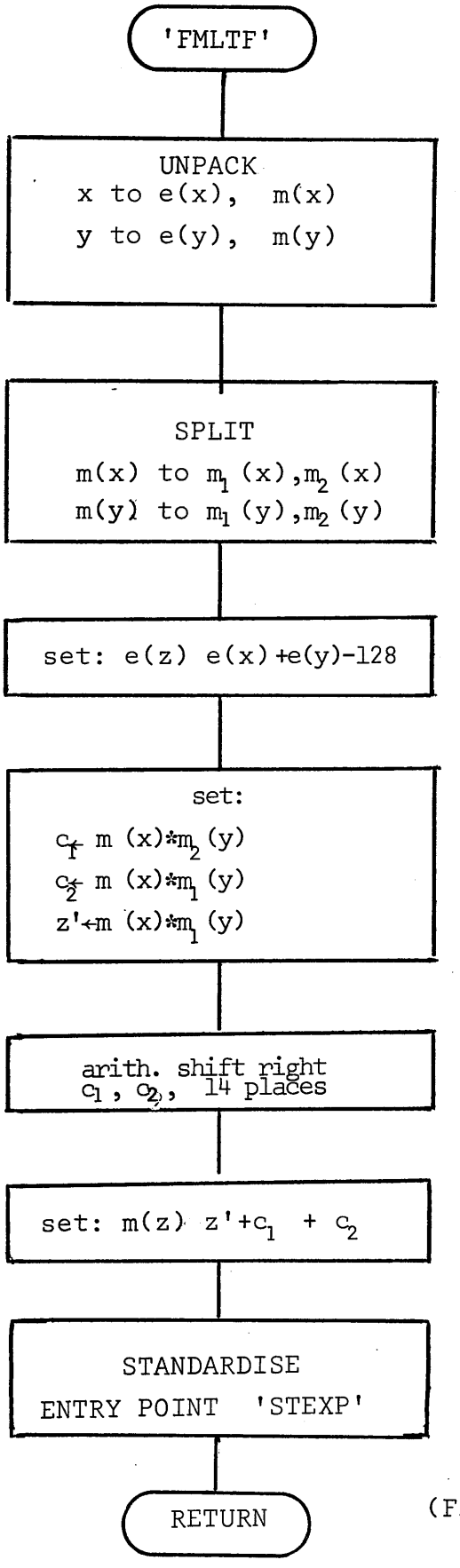
Suppose $m(x) = (80010400)_{16}$

then $m_1(x) = (8002)_{16}$ instead of $(8001)_{16}$

and $m_2(x) = (8200)_{16}$.

$m_2(x)$ is a 2's complement single length fraction. To get $m_2(x)$, the least significant word {of m(x)} is shifted right by one bit and then insert the sign bit according to the sign of m(x). As for $m_1(x)$, the most significant word {of m(x)} is increased by 1 {to give$(8002)_{10}$} If such adjustment in $m_1(x)$ is not carried out then

FLOWCHART: MULTIPLICATION
(FMLTF)



(Figure 9)

$$(80010000)_{16}$$
$$+\ (FFFF0400)_{16}$$
$$(80000400)_{16} \neq m(x)$$

If the least significant word of m(x) is zero, then $m_1(x)$ is returned as $(8001)_{16}$ and $m_2(x)$ as $(0000)_{16}$

Note that the procedure of splitting is only valid if the floating-point numbers are in the standard format. If not, overflow may occur in forming $m_1(x)$. For example, m(x) = $(FFFFF000)_{16}$, following the procedure described for splitting, $m_1(x)$ is then equal to $(0000)_{16}$ and overflow occurs as sign bit changes from 1 to 0.

2.3.8 DIVIDE (FDIVF)

The flowchart for floating-point divide is given in figure (10) and the procedure follows closely to that given in section (1.5.6) in Chapter One. The fixed-point instruction used in Step (3) of the procedure is DIVE. Here a 31-bit dividend is divided by a 16-bit divisor to give a 16-bit quotient and a 16-bit remainder. As in multiply, we need to devise a procedure for this type of mantissa arithmetic in order to get 24 significant digits (bits) for the mantissa.

Consider the mantissa arithmetic. Let m(z) be the quotient, m(x) the dividend and m(y) the divisor. Similarly we split m(y) into two single precision fixed-point fractions (refer section 2.3.7). Then

$$m(z) = m(x)/\{m_1(y) + \varepsilon m_2(y)\}$$

where ε is the reciprocal of the word size,

i.e. $$m(z) = \frac{m(x)}{m_1(y)} \{1 + \varepsilon m_2(y)/m_1(y)\}^{-1}$$

$$= \frac{m(x)}{m_1(y)} \{1 - \varepsilon \frac{m_2(y)}{m_1(y)} + \varepsilon^2 \left(\frac{m_2(y)}{m_1(y)}\right)^2 + \cdots\}$$

Neglecting terms $O(\varepsilon^2)$

$$m(z) \simeq \frac{m(x)}{m_1(y)} \{1 - \varepsilon \frac{m_2(y)}{m_1(y)}\}$$

Now $m(x)/m(y)$ gives a 16-bit quotient and a 16-bit remainder.

Let $m_1'(z) = m(x)/m_1(y)$

$= m_1'(z) + r'(z)$ where $r'(z)$ is the remainder,

and $r(z)/m_1(y) = m_2'(z) + r''(2)$ where $r''(z)$ is the remainder,

then $\simeq m'(z) \simeq m'(z) + m'(z)$

$$m(z) \simeq m'(z) \{1 - \varepsilon m_2(y)/m_1(y)\}$$

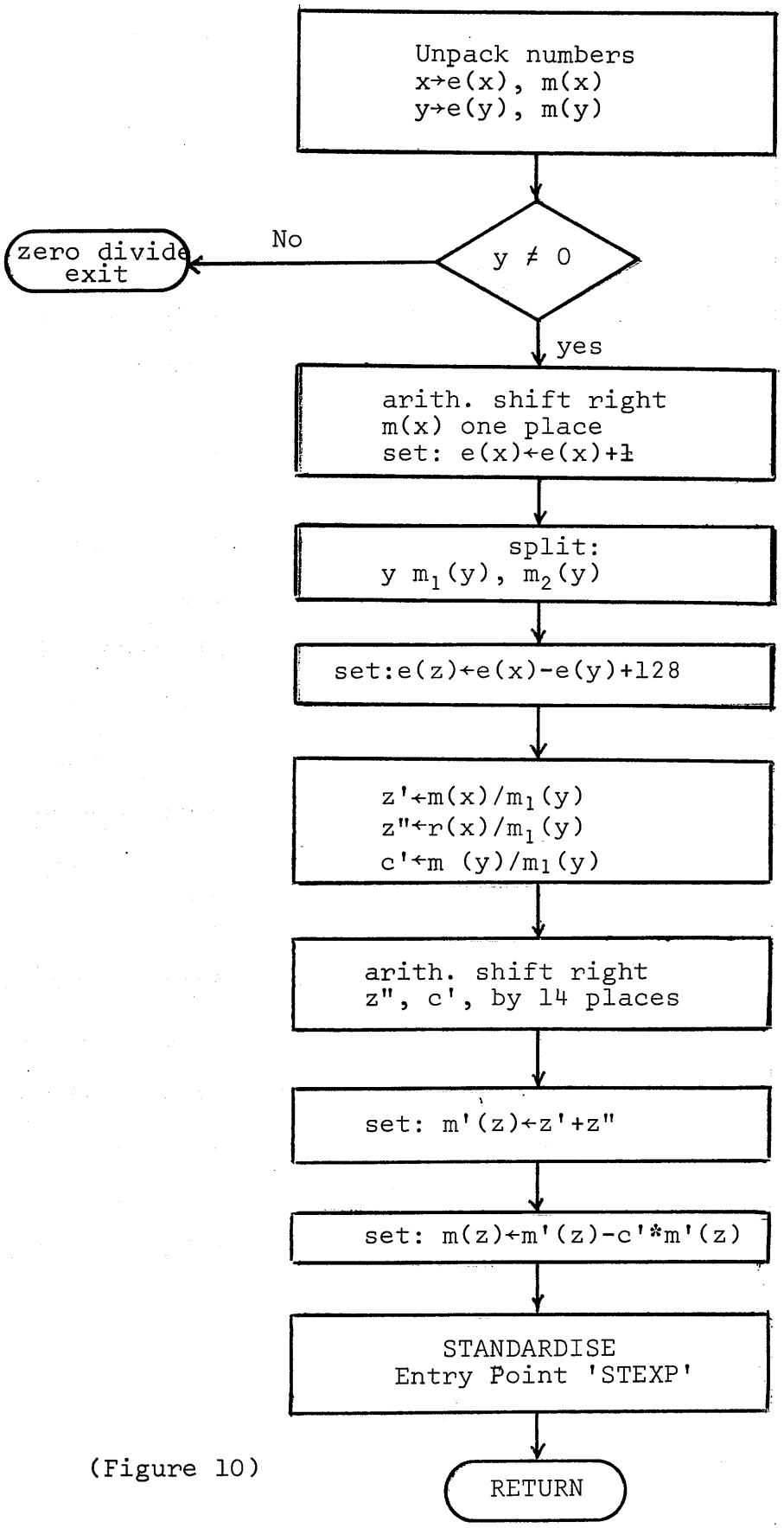
$$\simeq m'(z) - \frac{\varepsilon m_2(y)}{m_1(y)} m'(z)$$

Remark:

1. Care should be taken in the coding of the routine. Likely error will occur in the shift factor ε (reciprocal of word size).
2. The correction term $\frac{\varepsilon m_2(y)}{m_1(y)} m'(z)$ is computed by multiplying $m_2(y)$ and $m'(z)$ before division by $m_1(y)$. The result is then shifted right by the amount given in ε. If we were to perform division (i.e. $\frac{m_2(y)}{m_1(y)}$ before multiplication, then $m_2(y)$ is extended to a double precision fraction before division. The result obtained will be slightly different, since floating-point arithmetic is not

FLOWCHART: DIVISION

(FDIVF)



(Figure 10)

associative (refer section 1.8).

2.3.9 STANDARDISED (FSTND)

This routine is a straightforward adaptation of the algorithm given in section(1.5.5) of Chapter One. There are two 'entry points' in this routine. The entry point 'STEXP' (refer to FSTND in Appendix 1) is used to form a standardised floating-point number when the extended mantissa (32 bits) and the exponent are given. For non-standard floating-point numbers the entry point is 'FSTND'.

Remark:

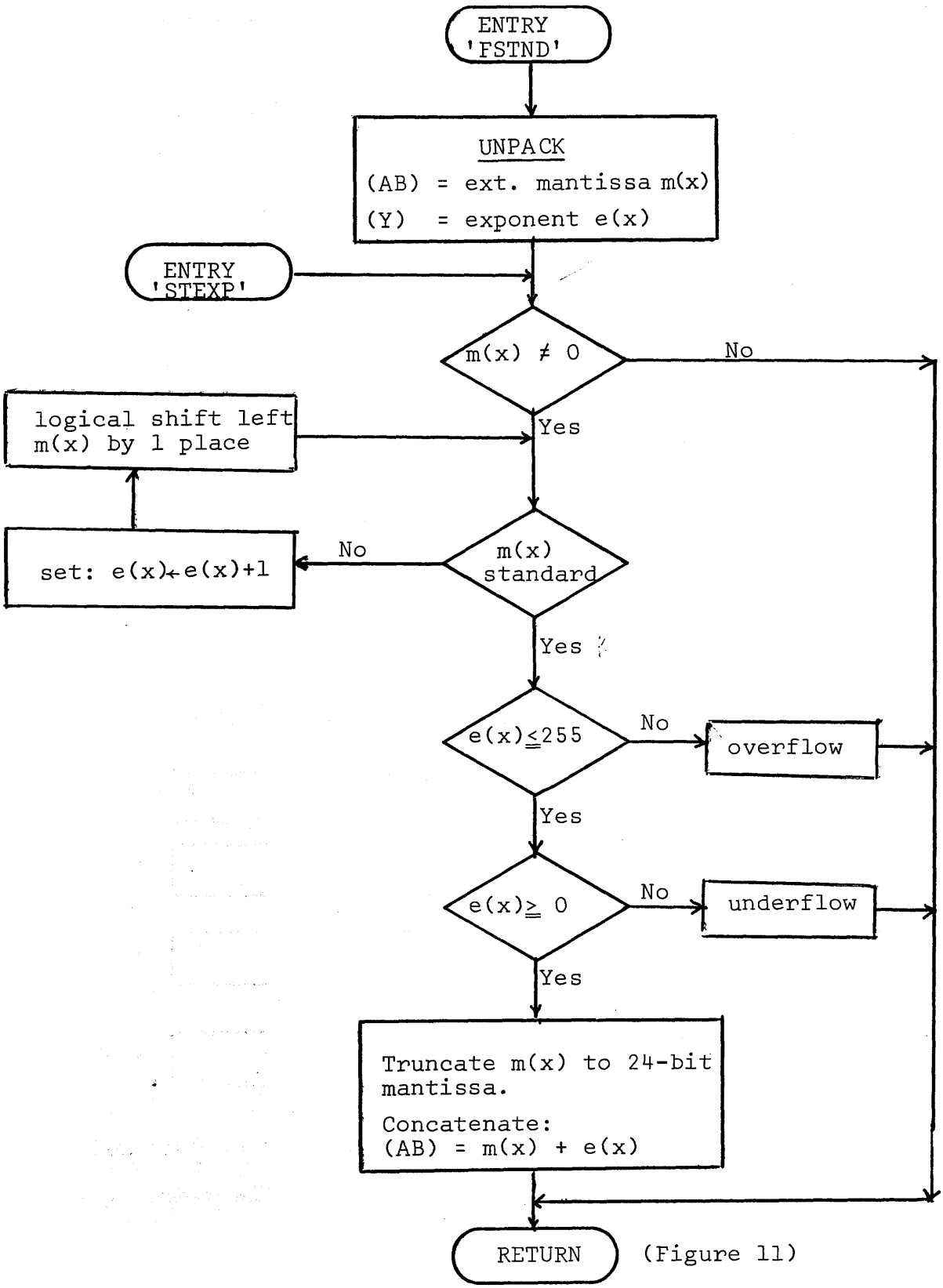
The 32-bit extended mantissa is truncated to a 24-bit fraction (instead of rounding as discussed in the first chapter) after standardisation. The exponent is tested for overflow and underflow before it is concatenated to form the standardised floating-point number.

The flowchart for the standardised routine is given in figure (11).

2.4 DOUBLE PRECISION ARITHMETICS

Two methods of implementing the double-precision floating point arithmetic are given in this section. Throughout the discussion, the standardised double length floating-point format proposed at Glasgow University (11) (refer to section 2.3.4) is assumed.

FLOWCHART: STANDARDISE
(FSTND)



(Figure 11)

2.4.1 METHOD 1

This is the conventional method as found in Knuth Difficulties arise in this method as the fixed-point instructions available do not give sufficient significant digits. We shall use the term "mantissa arithmetic' to denote operations performed on the mantissa. We need instructions capable of multiplying two 32-bit fixed-point numbers to give a 64-bit fixed-point product and also in division where a 32-bit dividend, divided by a 32-bit divisor, gives a 32-bit fixed-point quotient and a 32-bit fixed-point remainder. We term these operations as 'Long Multiply' and 'Long Divide' respectively.

We begin the discussion with the following proposed 64-bit fixed point format.

a) 64-bit fixed-point fraction.

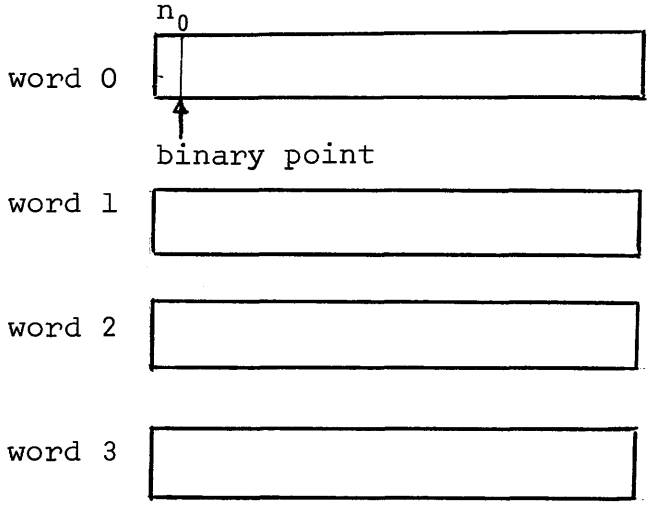


figure 12

The fraction is held as a 64-bit (4 words) 2's complement binary number with the assumed binary point immediately after bit 0 (as shown in figure 12). The fraction lies in the

range

$$-1 \leq f \leq 1-2^{-63}$$

b) fixed-point integer:

If binary point is assumed immediately after bit 63, the number represented is a fixed-point integer. The range cover by this format is $-2^{63} \leq i \leq 2^{63}-1$.

2.4.2 LONG MULTIPLY

The operands (32-bit number) are first split into two single precision fixed-point numbers such that $x = x_1 + \varepsilon x_2$ (refer to section 2.3.7). The procedure followed is the same as that described in the earlier section (where we split the extended mantissa into 2 16-bit fixed-point numbers). To obtain the 64-bit product, the following procedure is used (c.f. section 2.3.7).

Let x, y be two 32-bit fixed-point numbers and $x = x_1 + \varepsilon x_2$ where ε is the reciprocal of word size

x_1 being the most significant half

x_2 being the least significant half

Similarly, y is split into $y = y_1 + \varepsilon y_2$

$$x,y = (x_1 + \varepsilon x_2)(y_1 + \varepsilon y_2)$$

$$= x_1 y_1 + \varepsilon(x_1 y_2 + x_2 y_1) + \varepsilon^2 x_2 y_2$$

In this case term with ε^2 is not neglected as the term so formed may be significant. Following the same convention as that given for 'MLTA', the 64-bit product has bit 0 equal to bit 1 ($n_0 = n_1$). Overflow occurs if bit 0 is not equal to bit 1 (i.e. $n_0 \neq n_1$).

2.4.3 LONG DIVIDE

The procedures differ slightly for fixed-point fractionsand for fixed-point integers.

a) Fixed-point fraction:

Let x, y be two double length fixed-point fractions. We are required to find x/y. As x is a 32-bit fraction we only need to split y into two single precision fractions (c.f. section 2.3.7). We shall use the same algorithm given in section (2.3.8) (for floating-point division). Let $y = y + \varepsilon y_2$ where ε is the reciprocal of word size, then z, the approximate value for x/y, is given by

$$z \simeq \frac{x}{y_1} \{1 - \frac{y_2}{y_1} + O(\varepsilon^2)\}$$

The quotient z is a 32-bit fraction. The routine for long division does not return any value for the remainder.

b) Fixed-point integer:

We need to modify the procedure if the numbers are fixed-point integers. Similarly, the divisor y is split into two single length integers, say y_1 and y_2. If y_1 is equal to zero, division is the same as the instruction DIVE. We shall only consider the case when $y_1 \neq 0$

Let x be the 32-bit dividend and y be the divisor. Assume $y = \varepsilon y_1 + y_2$ where ε is the word size (i.e. 2^{15})

then $$x/y = \frac{x}{\varepsilon y_1 + y_2}$$

$$\simeq \frac{x}{\varepsilon y_1}\{1 - \frac{y_2}{\varepsilon y_1} + \frac{y_2^2}{\varepsilon y_1^2} \cdots\}$$

Let $\frac{x}{\varepsilon y_1} = z_1 + r$ where z_1 is the quotient (integer) and r_1 is the remainder.

$$\frac{x}{y} \simeq (z_1 + r_1)\left(1-\frac{y_2}{\varepsilon y_1}\right)$$

$$\simeq z_1 - \frac{z_1 y_2}{\varepsilon y_1} + r_1 - \frac{r_1 y_2}{\varepsilon y_1}$$

Since $\frac{x}{y}$ returns an integer, terms r_1, $\frac{r_1 y_2}{\varepsilon y_1}$ can be neglected since they are fractions (i.e. < 1).
The correction term $\frac{z_1 y_2}{\varepsilon y_1}$ is obtained by first multiplying r_1 and y_2 and then dividing by y_1. The result is then scaled to form a 32-bit fixed-point number. The 32-bit quotient is correct if either word 0 is equal to zero or $(FFFF)_{16}$. Any other patterns in word 0 indicate an overflow (or undefined). The 32-bit remainder may be obtained by multiplying the 32-bit quotient with the 32-bit divisor and subtracting this amount from x.

It remains to describe the addition (or subtraction) of two 64-bit fixed-point numbers (both fractions or both integers). Similarly, the operands are first split to form two 32-bit fixed-point numbers.

(i) Let x, and y be two fixed-point fractions and

$$x = x_1 + \varepsilon x_2$$

$$y = y_1 + \varepsilon y_2$$

where ε is the reciprocal of word size and x_1, x_2, y_1, y_2 are two's complement double length fixed-point fractions.

Then $x + y = (x_1 + y_1) + \varepsilon(x_2 + y_2)$

then $x + y = (x_1 + y_1) + \varepsilon(x_2 + y_2)$

(ii) If x and y are integers

then let $x = \varepsilon' x_1 + x_2$

$y = \varepsilon' y_1 + y_2$

where ε' is the word size (2^{31}) and x_1, x_2, y_1, y_2 are 32-bit fixed-point integers.

Similarly, $x + y = \varepsilon'(x_1 + y_1) + (x_2 + y_2)$.....(*)

Special care should be taken in evaluating the term $\varepsilon(x_2 + y_2)$ or $(x_2 + y_2)$ in equation (*). Overflow may occur when adding the terms x_2 and y_2.

FLOATING-POINT (DOUBLE PRECISION) ARITHMETICS

2.4.4 METHOD 1

With the availability of 'Long multiply', 'Long divide' and extended precision (to 64-bit) addition and subtraction we can follow the algorithms given in section (2.3.6) for single precision floating-point arithmetics to design the double-precision floating-point operations.

We do not go into the details of designing the routines as it would be the same as that described in the single precision routines. It should be pointed out that care needs to be taken in getting the 'reduced exponent' and in concatenating the mantissa and exponents to form the standardised double precision floating-point number. The 64-bit extended mantissa is truncated to 48 bits after standardisation (c.f. section 2.3.9).

2.4.5 METHOD 2 (Dekker's method)

The second method takes advantage of the algorithm

for exact addition, exact multiplication and exact division, as described in Chapter One. The procedures are given in Appendix (2). The algorithms make use of the floating-point (single-length) operations available and deliver the result as a double length floating-point number.

The accuracy of the algorithms depends mainly on how we split the double-length floating-point number to two single-length floating-point numbers. The theorem which follows gives the method by which a double precision floating-point number should be split.

THEOREM (2.4.6) (DEKKER)

Let R be the floating-point number system such that

$$R = \{x | x = m(x)\ 2^{e(x)},\ |m(x)| < 2^{t}\}.$$

If floating-point addition and subtraction are optimal, multiplication is faithful and t_1 and t_2 and C are defined by equations

$$t_2 = \text{entier}\left(\frac{t}{2}\right)$$

$$t_1 = t - t_2$$

$$C = 2^{t_1} + 1$$

then for all x in R, the following set of formulae

$$p = fl(x \times c)$$

$$q = fl(x - p)$$

$$h(x) = fl(q + p)$$

yields h(x) defined by the equation

$$h(x) = \text{round}\{m(x)2^{-t_1}\}2^{e(x)+t_1}$$

Proof:

If x = 0 then the theorem is obvious.

Assume $x \neq 0$ and floating-point numbers are normalised, i.e. $x = m(x)2^{e(x)}$

and $2^{t-1} \leq |m(x)| < 2^t$

since addition is optimal and given

$h(x) = \text{round}\{m(x)\ 2^{-t_1}\}2^{e(x)+t_1}$ which implies

$h(x) \in R(t_2)$ is contained in $R(t)$

we need only to show that $q + p \in R$ and

$q + p = \text{round}\{m(x)2^{-t_1}\}2^{e(x)+t_1}$

i.e. to prove

$$h(x) = q + p$$

Now $p = fl(x \times c)$

Let $p = m(p)2^{e(p)}$

obviously $e(p) = e(x) + t_1$ or $e(x) + t_1 + 1$

a) $e(p) = e(x) + t_1$

since $q = fl(x-p)$,

then $e(q)$ is equal either $e(p)$ or $e(p) - 1$.

If $e(q) = e(p) - 1$

then $|m(q)| \geq 2^t$ which is outside the mantissa range.

Hence $e(q) = e(p)$

and result follows.

b) $e(p) = e(x) + t_1 + 1$

Then obviously $e(q) = e(p) - 1$ or $e(p)$.

If $e(q) = e(p) - 1$

result follows.

If $e(q) = e(p)$ then we have

$$|m(q)| < |m(x)|/2 + 3/2$$

This is within the normalised mantissa range.

$2^{t-1} \leq |m(q)| < 2^t$ only if $|m(x)| = 2^t - \varepsilon$

where $\varepsilon = 1$ or 2

but then we obtain

$q + p = \text{round}\{m(x)\ 2^{-t_1-1}\}\ 2^{e(p)}$

$$= \text{round}\{m(x)\ 2^{-t_1}\}\ 2^{e(p)-1}$$

$$\text{i.e. } q + p = \text{round}\{m(x)\ 2^{-t_1}\}\ 2^{e(x)+t_1}$$

Hence $h(x) = q + p$.

From theorem (2.4.6) we have

$$t(x) = fl\{x-h(x)\}$$

and this is equivalent to

$$t(x) = x-h(x)$$

since subtraction is optimal and $t(x) \in R$.

Now given a double precision number x, we can split this into a pair of single-length floating-point numbers $\{h(x), t(x)\}$. It remains for us to give a formal definition of double precision floating-point number.

DEFINITION (DEKKER) 2.4.7

A double precision floating-point number is a pair (r, s) of single-length floating-point numbers $(r, s) \in R$ satisfying

$$|s| \leq |r + s| \frac{2^{-t}}{1+2^{-t}} \quad \ldots\ldots (2.4.8)$$

The value of the double precision number (r, s) is, by definition, equal to r + s.

This definition of double precision floating-point numbers fits in neatly to the proposed double precision floating-point format. The number representation in our system is a particular case of the system R where floating-point numbers are not standardised. Neglecting overflow and underflow, the theorems stated still hold. Also minor adaptation is required in the mantissa arithmetic. In Dekker's system, the mantissa is taken to be a fixed-point

integer, whereas it is defined as a fixed-point fraction in the computer (MULTUM) representation. The double precision floating-point number is split in the following manner. Word 0 and word 1 of the double precision floating-point number are taken to be h(x) (i.e. the head) and t(x) be represented in word 2, 3 of the floating-point number. We need only to put in the sign bit in bit 0 of word 2 to form t(x). If word 2, and word 3 are equal to zero t(x) is taken to be zero. Similarly, adjustment is required in forming h(x) if x is negative and the least significant mantissa is not zero (c.f. section 2.3.7). The adjustment is the same as that described earlier when we truncate the mantissa to a single precision number. In particular, if s = 0, we have a double-length number (r, 0). If addition is superfaithful then any pair (z, zz) obtained by performing an exact addition is a double precision floating-point number.

If condition (2.4.8) is replaced by

$$s \leq |r + s| C\, 2^{-t}$$

where C is some constant nearly equal to 1 (but greater than 1) then the pair (r, s) satisfying this inequality is defined as nearly double precision floating-point number. In particular, a pair of floating-point numbers (z, zz) obtained by exact multiplication is nearly double precision floating-point number since C is equal to $\alpha/(1+\alpha 2^{-t})$ in condition (1.9.15) in Chapter One.

2.5 DOUBLE PRECISION ADD/SUBTRACT

The double precision sum of two (nearly) double length floating-point numbers (x, xx) and (y,yy) is calculated as follows:-

2.5.1 ALGORITHM

1. The heads of x and y (viz. h(x) and h(y) are added exactly (refer Appendix 2). We assume $|x|>|y|$ in the algorithm. If $|y|>|x|$, the role of the terms x and y is interchanged.
   A double precision number (r, rr) is obtained using the algorithm for exact addition.
   i.e. r + rr = x + y.

2. To get the tail s, we perform the floating-point addition where
   $$s = fl\{(rr+yy) +xx\}.....(**)$$
   so that (r, s) approximates the sum of (x, xx) and (y, yy). Similarly, the role of the terms xx and yy is interchanged if $|xx|<|yy|$. This reduces the maximum error in (**) and ensures commutativity.

Note that floating-point (single precision) arithmetic assumes numbers to be in standardised form. Since single-length addition is 'superfaithful' (refer definition 1.8.7) the final exact addition of r and s transforms the approximate sum into a double precision floating-point number having the same value.

The pair (r, s) of floating-point numbers is concatenated to form the double precision floating-point number.

Care should be taken in setting up the reduced exponent in word 3. The procedures for double precision add is given as an ALGOL 60 procedure, add 2 in Appendix 2.

Similarly, the procedure for double-length floating-point subtract can be obtained. The calculations of double precision floating-point product and divide are straightforward coding of the algorithms given for exact multiplication and exact division respectively. Procedures in ALGOL 60 are given in Appendix 2 for multiply and divide.

In forming the product, a nearly double precision approximation (c, cc) of the required result is first calculated. The exact multiplication procedure is followed. The pair (c, cc) satisfies the condition $|s| \leq |r + s|\alpha 2^{-t}$ where α is the constant as stated in Definition (2.4.7).

Finally, an exact addition is performed to transform the result into a double precision floating-point number having the same value (refer to Appendix 2: ALGOL procedure multi, Mult 2).

2.5.2 Remarks

1. Cancellation may occur in forming r = fl(x+y), thus making |r| slightly less than |s|. If |r| = 0 we return the standardised value of s as the double-precision solution since in exact addition r + s is equal to s. Note, however, that when |r| ≠ 0 but less than |s|, and the

condition $e(r) \geq e(s)$ still holds, the role of r and s is not changed in the final exact addition.

2. To obtain the negative value of a double precision floating-point number we simply set x equal to zero and then perform the exact subtraction with y. This will give us the negative value of y provided that no overflow occurs in the process of subtraction. This method is not very efficient as it would involve a lot of unnecessary arithmetic operations. An alternative is an algorithm as described in the earlier section for single-precision floating-point numbers.

ERROR ANALYSIS

The error in the double-length addition is committed in calculating $s = fl\{(rr+yy) + xx\}$

If (x, xx) and (y, yy) are nearly double length numbers satisfying

$$|xx| \leq |x+xx| C_1 2^{-t} \quad \text{and} \quad |yy| \leq |y+yy| C_2 2^{-t}$$

where C_1 and C_2 are constants very near one (but greater than 1) then the error E of double length addition satisfies the relation

$$|E| \leq \{|x+xx|(1+C_1)+|y+yy|(1+C_2)\}2^{1-2t}.$$

Proof of this inequality is given by Dekker (6). Also, analysis of errors for multiplication and division are also found in (6).

2.6 CONCLUSION

The procedures given in Appendix 2 were not tested on the MULTUM ALP 2 computer as no floating-point operations were available then. If the arithmetic routines given in Appendix 1 were used in place of the hardware instructions, it would have involved a lot of procedure calls. This is very time consuming and inefficient. In addition, there is an uncertainty as to which double precision floating-point format will be adopted by the manufacturer.

As discussed in section (2.3.4), the floating-point double precision format with reduced exponent seems a better choice from the point of view of implementing Dekker's procedures. Both double precision formats may encounter early underflow. More risk is involved if the format proposed by I.C.S (refer section 2.3.3) were used. The double precision format in this case has a large exponent range (32768, 32767) and this may give overflow or underflow in the splitting stage where the double precision number is split to a pair of single precision floating-point numbers. Such early overflow will not occur in the other double precision format as both the single precision and double precision have the same exponent range. (128, 127). Splitting the double precision number to a pair of single precision numbers is easier. However, in both cases the smaller term (zz) is required in the standardised format and likely underflow may occur in the process of standardisation. Difficulties arise also in

concatenating the pair of single precision floating-point numbers to the double precision format. In the case of the format with reduced exponent, underflow may occur in the adjustment of the reduced exponent (to e(x)-23). However, this can be satisfactorily solved by clearing word 2 and word 3 of the floating-point number (i.e. set least significant mantissa and the reduced exponent to zero). This is fairly acceptable as the least significant mantissa would be out of range in this case.

Dekker's method seems more suitable for computers with large exponent range in the single precision format. This may prevent unnecessary underflow in the earlier stages of the calculation. The procedures have actually been tested on the Philip's Electrologica X8 computer at the Mathematical Centre, Amsterdam (6). The exponent range of the Philip's Electrologica X8 is $[-2048, 2048]$ (i.e. 12-bit for the exponent) and a 40-bit mantissa.

In the other method of implementing double precision arithmetic (section 2.3.3), both formats are likely to encounter the same sort of difficulties. Routines for 'Long multiply' (section 2.4.2) and 'Long divide' will be required for either format. In any case, the double precision multiply ('Long multiply') and double precision division ('Long divide') for integers are required for the Fortran compiler and the Pascal compiler, and it may seem more practical to have the double precision floating-point operations simulated in this conventional way. The

likely problem in this method will be similar to those routines written for single precision arithmetic.

The routines given in Appendix 1 were tested on the MULTUM computer. Random checks were made to test the solutions given by the hardware instructions and those given by the software. A bug was found in the hardware for floating-point addition and floating-point subtraction. Instead of returning the value $(4000\ 006A)_{16}$ for adding $(4000\ 0081)_{61}$ and $(8000\ 0180)_{16}$ or subtracting $(4000\ 0081)_{16}$ and $(7FFF\ FF80)_{16}$ the value $(0000\ 0000)_{16}$ was returned. (i.e. $(1+(1-2^{-15}))$ or $(1-(1-2^{-15}))$ give 0 instead of 2^{-15}). The software operations on these numbers returned the correct result. Floating-point premature overflows and inconsistency occur in the hardware operations for multiplication and division.

Floating-point premature overflows occur in multiplication when the unnormalised product has an exponent equal to the maximum allowable exponent plus one (i.e. $x = m(x)\ 2^{255+1}$ where $0 < m(x) < \frac{1}{2}$, and 255 is the biased form of the maximum exponent). For example, if $x = \frac{1}{2}\ 2^{255}$ and $y = \frac{1}{2}\ 2^{129}$ then $xy = \frac{1}{4}\ 2^{256}$. Exponent overflow is tested at this stage by the hardware, which obviously will give an overflow. If the exponent was tested after standardisation, the result $\frac{1}{2} \cdot 2^{255}$ is returned, which is allowed. The result of multiplication in this case forms a proper subset of the floating-point number. Such reduction in the range of the floating-point is hazardous

in computation. This, too, will greatly affect the implementation of double precision arithmetic by Dekker's method as overflow occurs in some intermediate computations and floating-point multiplication is not faithful. (Refer section 1.8.7). For details of hardware operations refer 'Specification of the ALP type 3' (13). To avoid such premature overflow either increase size of the exponent registers or perform testing to see if the mantissa is less than half before testing the exponent for overflow. The earlier case may be too expensive and the latter may have a heavy penalty in the efficiency of the operation.

Floating-point multiplication by hardware is not commutative. For example, if $x = \frac{1}{2}.2^{128}$ (i.e. $x = 1$) and $y = (-1).2^{255}$ (i.e. the largest allowable negative number) the product xy gives an overflow as y is first negated to give a positive value. Overflow occurs in negating the largest negative number. However the product yx will not give an overflow as only the second operand is tested for overflow if it is negative. In theory, the product xy is a floating-point number in the range of floating-point numbers. Such inconsistency may lead to erroneous results in the evaluation of power series. The hardware operations and those simulated by software are found to be consistent with each other (other than that mentioned earlier). Flowcharts for fixed integer, negate, etc. are given in Appendix 1. It is straightforward coding and no further discussion on these routines is given.

BIBLIOGRAPHY

1 Winograd, How fast can computer add? Computers and Computation. Reading from Scientific American (1971)

2 Ashenhurst, R.L. Number Representation and Significance Monitoring Mathematical Software (Ed. Rice) 1971, Academic Press

3 Knuth, E.D. The art of Computer Programming, Volume 2, "Semi-numerical Algorithms" 1969, Addison Wesley

4 Wilkinson, J.H. Rounding Errors in Algebraic Process. Her Majesty's Stationery Office (1963)

5 Garu, A.A. On a Floating-point Number Representation for Use with Algorithmic Language C.ACM 5, pg. 160-161

6 Dekker, T.J. A Floating-point Technique for Extending the Available Precision, Numerische Mathematik, 18, 1971, pg. 224-242

7 Ashenhurst, R.L. Unnormalised Floating-point Arithmetics J.ACM 6 (July 1959) pg. 415-428

8 Wadey, W.G. Floating-point Arithmetics J.ACM 7 (April 1960), pg. 129-139

9 Yardly D.T. Specification for the Alp Type 2 Information Computer System (I.C.S.) 2002-042

10 Ashurst, C.A. Usercode Language, Information Computer System, (I.C.S.) No. 2003-024

11 Glasgow University — Multum Hardware Memo. No. 2 (17/8/72). Fixed and Floating-Point Arithmetics

12 Naur, P. (ed.) — Revised Report on the Algorithmic Language. ALGOL 60 (1962)

13 Illing, D. — Specification of the Alp type 3 Information Computer System (I.C.S.) 2002-052

PART TWO

INTRODUCTION

Demands for higher quality in standard mathematical libraries have been building up. In recent years, programmes in these basic libraries have been subjected to thorough scrutiny and users expect the library to achieve maximum accuracy. By maximum accuracy we mean the results returned by the library to be accurate to the last digit. At this level of last digit accuracy, we regard the given argument value as exact and aim at producing an answer value that is the nearest in the given precision to the exact infinite precision answer (1). In most instances, this goal can be attained only by carrying out parts of computations with working precision higher than that of the library, especially when the relative accuracy of the result is very sensitive to the accuracy of the argument. In Chapter Three, we look into how the added accuracy helps to limit the accumulation of round-off errors, improving the probability of successful computation.

LAYOUT OF CHAPTER THREE

1. OBJECTIVE
2. CLASSIFICATION
3. ERROR ASSESSMENTS
4. MATHEMATICAL BACKGROUND
5. RANGE REDUCTION

In Chapter Four some of the basic library routines are discussed. The later part of the chapter gives a survey

of methods available in extending the library to include double precision routines. A brief discussion of the performance and testings of the library routines is also included.

The primary objective of this part of the thesis is to give a general survey of the methods available and the considerations required in the designing of a basic (mathematics) library. No attempt is made to prove any of the theorems quoted in Chapter Three and no proof is given for the polynominal approximations to the functions in Chapter Four. Literature regarding the proofs and theorems can be found in Hart (2), Fike (3), Snyder (4), Rice (5) and Achieser (6).

CHAPTER THREE

3.0 OBJECTIVES

Libraries of programmes of elementary functions are the basic structure for any Scientific System Library. Stringent requirements for this library must be met to satisfy the general user. Computer manufacturers usually supply these basic system libraries together with the processors. Unfortunately it cannot be said that all manufacturers supply noteworthy sub-routine libraries (14). H. Kuki (7) listed some of the important conditions that must be observed in order to produce a library that would satisfy most users all the time and all the users most of the time.

a) Reliability:

The meaning of the word is self-explanatory. Absolute reliability is expected from the libraries. This includes both numerical accuracy and adequate diagnosis of errors. A rigid accuracy standard which is commensurate with the precision of the given computer must be maintained. Acceptable answers must be returned for all legitimate arguments and mark all other arguments. Numerical accuracy is normally attained at the expense of speed and storage requirements.

b) Domain:

The legal domain of function routine defined should include virtually all those arguments whose function values are representable in the number system of the given computer.

For those arguments which are outside the legal domain, proper diagnostics should be given.

c) Speed:

Optimal speed under the above constraints should be arrived at. This largely depends on the hardwares available and the choice of programming languages.

d) Size:

When all the above requirements are met, we are left only to scale down the size of the programme. It can be said that the smaller the better.

In general, Accuracy-Efficiency-Space conventions are used as the general guideline for designing library routines. Finally, we have to perform extensive testings of the routines to certify that the routine is creditable. For example, 'Random Test' to determine the reliability may be used. Unfortunately, this is not sufficient to ensure that routines are one hundred percent reliable (8). The best method to ensure reliability is for the coder to completely document his methods and for the certifier to check on the method and code, and then devise test cases which are likely to cause trouble.

3.1 CLASSIFICATION OF ROUTINES

Function routines are classified into three categories, namely, primary, secondary and management (14).

3.1.1 PRIMARY ROUTINES

These are the basic building blocks of any library.

The routines are completely self-contained and rely upon no other routines for computation. Instead, these routines are frequently called upon by other routines to perform critical computations. Examples of primary routines are routines for sine, cosine, logarithm, and exponential.

3.1.2 SECONDARY ROUTINES

Secondary routines do part of the computation themselves but rely upon other routines (e.g. primary routines or other secondary routines) for some of the computations. Typical routines for the inverse trigonometric functions and some hyperbolic functions are among the secondary routines in the library.

3.1.3 MANAGEMENT ROUTINES

These routines merely manage the flow of information from one routine to another. Computations are done by calling on the primary and secondary routines in the library Routines for exponentiation and for certain hyperbolic functions and complex functions are frequently in this category.

3.2 THE CHOICE OF PROGRAMMING LANGUAGE

To produce optimal programmes in regard to the conditions described earlier, it is necessary to make the most out of the peculiarities of the computer for which the library is designed. Accuracy and domain require-

ments are stated in terms of the machine. In addition, the reduction algorithms of several elementary functions are intimately related to the internal number representation. This leads to the choice of the Assembler code for basic system libraries. H. Kuki (7) concludes that in the present state of compiler art, codes generated by computers cannot compete with the assembler code in terms of economy of space and speed. The timing figures and storage requirements shown in Table 1 are excerpted from H. Kuki's paper

Three Fortran codes were prepared and execution times cited are for the I.B.M. 360/65G computer. Codes for the exponential, logarithm, sine/cosine sub-programmes are aimed at the basic accuracy of 10^{-7}. For details of the coding of these routines, refer to (7). The compiled code excludes the routines that are referred to.

The results only iterate the demands for the Assembler Codes be used for writing basic library routines.

TABLE 1 Comparison of the Timing and Storage between Fortran Code and Assembler Code

FUNCTION	FORTRAN CODE STORAGE (IN BYTES)	TIMING (μ SEC.)	ASSEMBLER CODE STORAGE(IN BYTES)	TIMING (μ SEC.)
Exponential Exp(x)	452	128	184	86
Logarithm A LOG (x)	458	291	184	83
SINE/COSINE	712	169	200	76

3.3. SYSTEM SPECIFICATIONS

Those features that either enhance the capabilities of the host system or features that are required by the host system should be provided. Arguments that are outside the specified domain should have proper diagnostics. In general programmes in a standard library should be equipped to handle any input argument that is capable of being produced by the host system. For example, if the host system only works with normalised floating-point numbers, any input arguments that are unnormalised may produce erroneous answers. Also programmes should behave consistently with the conventions of the host system. For example, treatment of underflows and overflows should be appropriately dealt with. Users should have an alternative as to whether to be informed of occurrence of underflow or not.

3.4 STANDARD REFERENCE FOR ACCURACY

Accuracy is defined here as the measure of deviation of the computed answer from the exact one. Assume that the given argument value is exact, then the infinite precision function value corresponding to this argument value is taken to be the exact answer. By exact argument, we mean that value for which the argument in its machine representation stands as it is passed on to the sub-routine. No allowance is made for minimal rounding error, conversion errors and accumulation of errors from prior processing.

3.4.1 EFFECT OF AN ARGUMENT ERROR

There are two major sources of error associated with any function value.

Transmitted error:

This is an error due to a small error in the argument. Let x be the argument and y = f(x) be the exact function value. Also write Δx and δx as the absolute and relative error of x inherited from prior computation, respectively. If Δy and δy are respectively the absolute and relative error in y, and function f(x) is a differentiable function, then

$$\Delta y = \frac{df}{dx} \Delta x$$

and

$$\delta y = \frac{\Delta y}{y} \simeq \frac{dy}{y}$$

$$= \frac{f'(x)}{f(x)} dx \simeq \frac{f'(x)}{f(x)} \delta x$$

i.e.

$$\delta y \simeq \frac{x}{y} \frac{df}{dx} \delta x.$$

The transmitted error δy depends solely on the inherited error δx and not on the sub-routine.

3.4.2 GENERATED ERROR

This type of error is generated by the sub-routine. It includes the error due to approximation of the function f by an arithmetically specifiable function ϕ, as well as error due to the round-off characteristics of the machine. In particular, it includes the error due to the inexact representation of constants as machine words.

Since sub-routines have no control over the inherited error, they should then be designed to minimise the

generated error. The next few sections survey the methods available in approximating the function f so that generated error is minimal.

Remark:

a) We shall use the same notation f to denote both the function and the infinite expansion (approximation) of this function since mathematically they are equivalent.

Let ϕ be the finite approximation of f. If x is the argument value that is passed to the sub-routine (i.e. $x^*-x = \Delta x$ where x^* is the exact argument) then the generated error will be $f(x)-\phi(x)$ (c.f. Ashenhurst (4)) or $f(x)-\phi(x)/f(x)$ depending on the methods of error measurement (c.f. Fike (3)).

b) From equation (3.4.3)

$$\delta y = xf'(x)/f(x)\delta x \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad 3.4.3$$

the function corresponding to the argument x is said to be unstable if $xf'(x)/f(x)$ is very large.

3.4.4 TECHNIQUES OF REDUCING GENERATED ERRORS

Generated errors of the type mentioned above may be reduced with some additional codes. The basic technique is to use guard digits at crucial points in the computations. Fixed-point computation is used profitably when the precision of the fixed-point representation is several digits longer than that of the floating-point. For single precision sub-routines, a limited use of double precision computations

allows a very accurate computation of the reduce argument in the reduction stage. However, in double precision routines such back-up precision (multiple precision) is not usually available. If available, the time penalty for using the multiple precision calculations is very heavy. It would be too expensive to carry out the entire calculation in multiple precision. Instead, a certain strategy is followed. The crucial step is usually located and computation at this stage is carried out (in extra precision) to obtain a better approximation of the exact value. This technique may turn out to be far less expensive (in terms of speed and storage) than to carry out a full precision calculation on every step.

Listed in Table (2) (given in (2)) are certain techniques that may be useful in the writing of an efficient code.

TABLE 2

NAME	DESCRIPTION
Fixed-point Arithmetic	As described in section (3.4.4) to provide good error control.
Equivalent Operations	Certain operations may be replaced by those arithmetic or logical operations that are more efficient. For example, if division is slower than multiplication, it may be desirable to store the reciprocal of a constant and use multiplication rather than division. Also in

	cases of multiplication or division by 2, in fixed-point it is better to make use of the shift operations. In case of floating-point arithmetic, replace multiplication and division by addition or subtraction involving the exponent part of a floating-point number.
Variable Timing	The time of evaluation of a power series may be shortened by forcing coefficients to assume a form which is exactly representable on a computer by a few digits.
Shared Storage for Instructions	Different sub-routines may share common sections of code or tables of constants. It would be more economical to combine these routines in one multiple entry suh-routine. Example of shared storage routines is the sine and cosine routines.
Inner Loop Efficiency	Remove from inner loops all operations which can equally well go outside the loop.

3.5 THEORETICAL BACKGROUND

The computation of a mathematical function consists, in general, of two stages; the reduction stage and the approximation stage. The basic stage into which the argument is reduced must be such that, within this range, one has an approximation algorithm that is both efficient and stable. Our accuracy goal is to keep the maximum relative error well within the range of the last digit value of the working precision.

Among polynominal (or rational) approximations of a given degree there is one that minimises the maximum error in the given range. In the next section we shall give a brief discussion of this type of approximation, namely minimax approximation. Several algorithms are available to us to determine coefficients of such approximations. We shall confine ourselves to Remez's method and Chebyshev interpolation in determining the coefficients.

No attempts will be made to prove any of the theorems given in the next few sections on polynomial approximations. Most of the tables and data given are excerpted from Hart (2).

3.5.1 MINIMAX POLYNOMIAL APPROXIMATIONS

In the preceding sections we have discussed the nature and requirements of basic library sub-routines. Also we have considered factors which will influence the results. In the sections that follow we give a survey of a type of optimum approximation that minimises the generated error.

Instead of permitting the search for optimal approximations to range over the class of all possible algorithms for a given machine, we will restrict the search to polynomial approximations and the rational approximations.

Throughout the discussion, we will use f(x) to denote the function to be approximated, and $[a, b]$ to denote the approximation interval. The function f(x) is assumed to be continuous in this interval.

THEOREM 3.5.2 (Chebyshev's theorem on polynomial approximation)

Let u(x) denote a function continuous in the closed finite interval $[a, b]$ and let v(x) denote a function continuous and non-zero in $[a, b]$. Let V_n denote the set of polynomials of degree n. There exists a unique polynomial $P_n^*(x)$ in V_n such that

$$\max_{a,b} \left| \frac{P_n^*(x)}{v(x)} - u(x) \right| = \min_{P_n(x) \in V_n} \max_{[a,b]} \left| \frac{P_n(x)}{v_n(x)} - u(x) \right|$$

Let $P_n(x)$ denote a polynomial in V_n. Then $P_n(x)$ is $P^*(x)$ if and only if there exist $N \geq n+2$ points in $[a, b]$

$$x_1^* < x_2^* < x_3^* \ldots < x_N^*$$

such that

$$\frac{P_n(x_k^*)}{v(x_k^*)} - \mu(x_k^*) = (-1)^k \mu^* \quad \text{for } k=1, 2, \ldots N$$

where

$$|\mu^*| = \max_{[a,b]} \left| \frac{P_n(x)}{v(x)} - u(x) \right|.$$

Proof of the theorem can be found in Achieser (1956) (7).

Remarks:

(i) If $(x)=1$, and $u(x)=f(x)$,

then function $\frac{P_n(x)-u(x)}{\nu(x)}$ becomes the absolute function

$$P_n(x)-f(x).$$

In this case the theorem asserts that there exists a unique polynomial $P_n^*(x)$ of degree $\leq n$ that approximates $R(x)$ with minimax absolute error in $[a,b]$. Also the maximum discrepancy between f and P_n^* must occur with alternating sign at n+2 successive points of the interval. Finally, there must exist points

$$a \leq x_0^* < x_1^* < \ldots \leq b \text{ such that}$$

$$P_n^*(x_i^*) - f(x_i^*) = f(x_{i+1}^*) - P_n^*(x_{i+1}^*) = \pm||P_n^* - f||$$

(ii) With $(x)=f(x)$ and $u(x)=1$, and $f(x)\neq 0$ in $[a,b]$

then $\frac{P_n(x)}{\nu(x)} - u(x)$ becomes relative error function

$$\frac{P_n(x)-f(x)}{f(x)} .$$

Similarly as in (i) there exists a unique polynomial $P_n^*(x)$ of degree $\leq n$ that approximates $f(x)$ with maximum relative error in $[a, b]$ also there exist n+2 extreme points in a, b such that

$$(P_n^*(x)-f(x))/f(x) = (f(x)-P_n^*(x))/f(x)$$

if $f(x)=0$ for $x=\alpha$, then approximate does not hold unless the following conditions are satisfied.

1. The only point in $[a, b]$ at which $f(x)=0$ is $x=\alpha$
2. $f(x)/(x-\alpha)$ is non zero in $[a, b]$

This implies that $\lim_{x\to\alpha} f(x)/(x-\alpha)$ exists and we define $f(x)/(x-\alpha)$ to have that limit as its value at $x=\alpha$.

Therefore

$$P_n(x) = (x-\alpha)P_{n-1}(x)$$

where $P_{n-1}(x)$ is a polynomial of degree $\leq n-1$.

Also

$$P_n^*(x) = (x-\alpha)P_{n-1}^*(x)$$

and $P_n^*(x)$ approximates $f(x)/(x-\alpha)$ with minimax relative error in $[a, b]$.

Note: 1. The polynomial of degree $\leq n$ that approximates $f(x)$ with minimax absolute error in $[a, b]$ is, in general, not the same as the polynomial of degree $\leq n$ that approximates $f(x)$ with minimax relative error in $[a, b]$.

2. The minimax polynomial approximation is dependent on the integer n and the interval $[a, b]$. In general, changing either one will give a different approximation to a function. In fact, there is no general way to have one common minimax polynomial approximation to a function for different intervals or of different degrees. In other words, selecting an optimal approximation for a function in one computer may not necessarily give an optimal approximation in another machine as different machines have different number representation.

3. A minimax polynomial is an even/odd function if it is a minimax absolute error or minimax relative error approximation to an even/odd function in a symmetrical range (i.e. in interval $[-a, a]$).

Proof:

Suppose f(x) is an odd function and $P^*(x)$ is the unique polynomial of degree $\leq n$ that approximates f(x) with minimax relative error in $[-a, a]$,

Then

$$\max_{[-a,\,a]} \left|\frac{P_n^*(x)-f(x)}{f(x)}\right| = \max_{[-a,\,a]} \left|\frac{P_n^*(x)-f(-x)}{f(-x)}\right| .$$

Definition of odd function implies

$$f(x) = -f(-x)$$

$$\max_{[-a,\,a]} \left|\frac{P_n^*(x)-f(x)}{f(x)}\right| = \max_{[-a,\,a]} \left|\frac{P_n^*(-x) + f(x)}{-f(x)}\right|$$

$$= \max_{[-a,\,a]} \left|\frac{-P_n^*(-x) - f(x)}{f(x)}\right|$$

since $P^*(x)$ is unique

$$P^*(x) = -P_n^*(x)$$

which implies that $P_n^*(x)$ is an odd function-

3.5.3 NEARLY MINIMAX POLYNOMIAL APPROXIMATION

The theorem given below gives an estimate of the maximum absolute error in a minimax absolute approximation or an estimate of the relative error in the minimax relative approximation. The usefulness of this lies in the fact that it makes it possible to obtain knowledge of the accuracy of a minimax polynomial approximation without actually having to derive the polynomial itself.

THEOREM 3.5.4

Let $P_n(x)$ denote a polynomial of degree $\leq n$ and let $x_1, x_2, \ldots, x_{n+2}$ denote points such that

$$a \leq x_1 < x_2 < \ldots < x_{n+2} \leq b.$$

If $P_n(x_k)/v(x_k) - u(x_k) = (-1)^k \mu_k$, k=1, 2, ... n+2

where μ_1 , μ_2 , ..., μ_{n+2} denotes non-zero numbers having like signs.

Then

$$\min_{1\leq k\leq n+2} |\mu_k| \leq \max_{a,\ b} \left| \frac{P_n^*(x)}{v(x)} - u(x) \right|$$

(Refer Achieser (1956) for proof of theorem (6), Rice (5)).

Remark:

The theorem is interpreted in terms of minimax absolute error if $u(x)=f(x)$ and $v(x)=1$, in terms of minimax relative error if $u(x)=1$ and $v(x)=f(x)$.

The theorem implies that if the error function for an approximation alternates in sign at n+2 points in $[a, b]$, the magnitude of the maximum error for the corresponding minimax approximation is bounded below by the smallest of the error magnitude at the n+2 points.

3.5.5 REMEZ's SECOND METHOD

The algorithm of Remez exploits Chebyshev's Theorem for minimax polynomial approximation. Two methods for Remez's procedure will be considered, namely Remez's method for minimax absolute error and Remez's method for minimax relative error.

1. For minimax absolute error

Let $P_n(x)$ be the polynomial that approximates $f(x)$ with minimax absolute error in $[a, b]$, and

$$P_n(x) = a_0 + a_1 x + \ldots + a_n x^n .$$

From Chebyshev's Theorem (3.6.2), the standard error function processes exactly n+2 critical points in $[a, b]$,

say, x_k, $k=1, 2, \ldots, n+2$ such that

$$a = x_1 < x_2 < \ldots < x_{n+2} = b$$

and that

$$P_n(x_k) - f(x_k) = (-1)^k \mu \ , \ k = 1, 2, n+2, \ldots \quad (*)$$

where

$$|\mu| = \max_{[a, b]} \left| P_n(x) - f(x) \right|$$

The coefficients a_0 , a_1 , a_2 , ..., a_n are obtained by solving the system of equations (*), but the critical points are unknown. To obtain the coefficients for the minimax polynomial the following iterative method is followed.

1. Initially, select n+2 numbers x_k , k=1, 2, ..., n+2 such that

$$a = x_1 < x_2 < \ldots < x_{n+2} = b$$

2. Solve the set of equations

$$P_n(x_k) - (-1)^k \mu = f(x_k)$$

for the coefficients of $P_n(x)$ and μ.

3. Substitute the calculated values of a_0, a_1, ..., a_n and $P_n(x)$ and then locate the extreme point for the standard error function $\{P_n(x) - f(x)\}$ in $[a, b]$. Assume there are exactly n+2 extreme points, y_k, k=1,..., n+2, including a, and b such that

$$a = y_1 < y_2 \ \ldots \ y_3 < \ldots < y_{n+2} = b.$$

3. Replace x_k by y_k for k=1, 2, ... n+2 and repeat the sequence of steps given above beginning with 2.

It can be proved that x_k converges to x'_k and μ converges to μ' ($\neq 0$) and a_k converges to a'_k for any

starting value in step (1). (Refer Veidinger (1960) or Rice (1967) (9), (5) for proof of theorem). It is known (Veidinger (1960) that if f" exists and is continuous and if $|P_n - f|$ achieves its maximum at the end points and at exactly n other points, then the maximum deviation in Remez's algorithm converges quadratically to their infimum $||P_n - f||$.

Remark:

The system of linear equations (1) and (2) tends to become ill-conditioned as n increases. Modification to the method of the algorithm or the use of high precision arithmetic can overcome this problem.

3.6 FUNDAMENTAL PROPERTIES OF CHEKYSHEV POLYNOMIALS

The Chekyshev polynomial of degree n, $T_n(x)$ is defined recurringly for each non-negative integer n by the equation

$$T_n(x) = \text{Cos}\ (n \cdot \text{arc Cos}\ x) \quad \text{for } -1 \leq x \leq 1 \quad \ldots (3.6.1)$$

where $x = \text{Cos}\ \theta$.

The Chekyshev polynomials have a number of interesting and useful properties which can be derived from the Definition (3.6.1). Among these are the following:

1. $T_n(x)$ is a polynomial of degree n in x. If n is even $T_n(x)$ is an even polynomial; if n is odd, $T_n(x)$ is an odd polynomial.
2. The coefficient of x^n in $T_n(x)$ is 2^{n-1}.
3. $T_n(x)$ has exactly n real zeros in the interval $[-1, 1]$.

From Definition (3.6.1) these zeros are located at

$$x_j = \text{Cos}\frac{2j+1}{n}\frac{\pi}{2},\ j=0,\ 1,\ 2,\ \ldots n-1$$

4. $|T_n(x)| \leq |\ ,\ -1 \leq x \leq 1$ for all n.

5. $T_n(x)$ $(n>0)$ attains its bounds ± 1 alternatively at the points

$$x_j = \text{Cos}\ \frac{j\pi}{n}\ ,\quad j = 0,\ 1,\ 2,\ \ldots,\ n$$

$$T_n(x_j) = (-1)^j$$

6. Minimax property

Let $P_n(x)$ be any polynomial of degree n with leading coefficient unity. Then

$$\max_{-1\leq x\leq 1} |2^{1-n}T_n(x)| \leq \max_{-1\leq x\leq 1} |P_n(x)|$$

No attempts will be made to verify the properties stated. (For proofs, refer (5)).
These properties may be used to find polynomial approximations for function f(x) directly or indirectly.

3.7 CHEBYSHEV SERIES.

If the function f(x) has a continuous first derivative in $[-1, 1]$, then it processes a Chebyshev series expansion

$$f(x) = \sum_{k=0}^{\infty}{}' a_k T_k(x) = \tfrac{1}{2} a_0 T_0(x) + a_1 T_1(x) + a_2 T_2(x) + \ldots$$

which converges uniformly and absolutely in $[-1, 1]$ where the coefficients in the series are given by

$$a_k = \frac{2}{\pi}\int_{-1}^{1} f(x)\ T_k(x)\ (1-x^2)^{-\frac{1}{2}}\ dx$$

The infinite series is normally truncated to a finite

series and is an excellent means of obtaining near minimax approximations to the function f(x) in the range $[-1, 1]$.

3.8 POLYNOMIAL APPROXIMATION METHODS

Various methods are available to approximate a function such that the approximation is a nearly minimax approximation. Methods like truncation of power series, method of economisation, truncation of Chebyshev series and Chebyshev interpolation can be used to obtain polynomial approximations for the function f(x). Literature regarding these methods is readily available. (Refer (2), (3), (5)).

A brief description of Chebyshev's interpolation method will be given here as we will be using this type of approximation method in our discussion in Chapter Four.

3.8.1 CHEBYSHEV INTERPOLATION

Suppose f(x) and $f^{n+1}(x)$ (i.e. the function and the (n+1)st derivative) are continuous in the interval $[-1, 1]$ and $x_1, x_2, \ldots x_{n+1}$ are n+1 distinct numbers in $[-1, 1]$, then there exists a unique polynomial of degree $\leq n$

$$P_n(x) = a_0 + a_1 x + \ldots + a_n x^n$$

such that $P_n(x_k) = f(x_k)$ for k=1, 2, ..., n+1

The coefficients $a_0, \ldots, a_n$ can be obtained by solving n+1 linear equations

$$a_0 + a_1 x + \ldots + a_n x_k^n = f(x_k), \quad k=1, 2, \ldots, n$$

for the n+1 unknowns. $x_1, x_2, \ldots, x_{n+2}$ are called the nodes of the approximation. If the zeros of the Chebyshev polynomial $T_n(x)$ are taken to be the nodes of the approximation, the method is then called Chebyshev interpolation. The zeros for $T_{n+1}(x)$ is given by

$$x_k = \text{Cos}\frac{(2k-1)}{2(n+1)}\pi \quad k=1, 2, \ldots, n+1.$$

Remarks:

1. If f(x) is an even function, an approximation $P_n(x)$ obtained by Chebyshev interpolation is also an even function. Likewise, if f(x) is odd, $P_n(x)$ is odd.

2. Let $P_n(x)$ be a polynomial of degree $\leq$ n determined by Chebyshev interpolation as the approximation to a function f(x) in $[-1, 1]$, and given that

   $$P_n(x) = \tfrac{1}{2}b_0 T_1(x) + b_1 T_1(x) + \ldots + b_n T_n(x)$$

   then the coefficients b_p, p=0, 1, 2, ..., n can be computed by means of the formula

   $$b_p = \frac{2}{n+1} \sum_{k=1}^{n+1} f\left(\text{Cos}\frac{(2k-1)\pi}{2(n+1)}\right) \text{Cos}\frac{(2k-1)}{2(n+1)} p\pi.$$

3.8 RANGE REDUCTION

The computation of basic library functions involves a reduction of the argument to some primary interval, followed by the evaluation of an approximation to the function over that interval. In some cases, like sin(x),

ln x, there are no practical useful approximations to these functions for large ranges. It is, therefore, essential to be able to reduce the argument range over which such functions must be approximated.

It can be said that the quality of the library routines hinges upon the care used in the argument reduction stage. The main objective of the range reduction is to approximate the function with a minimum number of terms and also to avoid the occurrence of singularities. Suppose, for example, the function arc tan (x) is approximated by a polynomial of degree n, say $P_n(x)$ and that we require x in the range $[a, b]$. In Table (3) we can see that, for a given maximum relative error, the number of terms increases for increasingly large approximation intervals. The values in the table are excerpted from Hart (2).

However, in some cases, range reduction does not improve the accuracy in the approximation, for example, approximations involving the Gamma functions. In some applications, the logical approximation range is small enough for no range reduction, for example, if the function sinh (x) is computed with the aid of the identity $\sinh(x) = \frac{1}{2}(e^x - e^{-x})$, and approximation of e^x in the range $[-\frac{1}{2}\ln 2, \frac{1}{2}\ln 2]$. The evaluation of sinh (x) using this procedure is very unsatisfactory for very small argument. When x is nearly 0, e^x and e^{-x} are near 1 and cancellation occurs in the evaluation.

TABLE (3) Comparison of Range and Number of Terms Required to Achieve that Accuracy for Arc Tan (x)

[a b]	MAXIMUM RELATIVE ERROR		NO. OF TERMS n
$[0, \tan \frac{\Pi}{32}]$	.3987	10^{-8}	2
$[0, \tan \frac{\Pi}{16}]$	.1996	10^{-8}	3
$[0, \tan \frac{\Pi}{12}]$	.1121	10^{-9}	4
$[0, \tan \frac{\Pi}{8}]$	.1197	10^{-9}	5
$[0, \tan \frac{\Pi}{4}]$	.1383	10^{-8}	9

Segmented approximations lead to greater accuracy. However, we may need more storage space to hold the constants for each of the segmented intervals. Logical tests will be required to determine which segment (interval) the argument belongs to. Functional properties like periodicity, symmetry, addition formulae and recurrence relations sometimes allow some range reduction without the use of segmented approximations. For example, in approximating the function sin(x), the periodicity helps to reduce the complexity of range reduction (refer section (4.2.2) Chapter 4).

In all the routines discussed in Chapter Four, range reductions are carried out. The error in the reduction process will be carried forward as argument errors

(refer section 3.4.1). Extreme care is required to prevent this. As discussed in section (3.4.4) extra precision arithmetic may be required to calculate the reduced arguments. One method is by phase reduction where only critical steps are carried out in double precision arithmetic (refer (7)).

3.9 POLYNOMIAL EVALUATION METHODS

The technique used to evaluate a polynomial approximation affects the speed of a function evaluation routine. The most common and frequently used polynomial evaluation method is the technique called 'nest multiplication'. This method is simple to perform (code) and also it is numerically reliable. For example, to evaluate a fourth degree polynomial

$$P_4(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

we express the polynomial in the form

$$P_4(x) = ((a_4 x + a_3)x + a_2)x + a_1)x + a_0 .$$

In general, a n^{th} degree polynomial will require n multiplication and n addition to evaluate the polynomial.

This technique of evaluation will be extensively used in Chapter Four. Other methods such as economical evaluation will not be considered, as evaluating an approximation polynomial with the aid of an economical evaluation method can often be unsatisfactory numerically.

Rounding errors tend to propagate in this method even though the evaluation is faster than evaluation by nested multiplication (as economical evaluations require less multiplication). Details of polynomial evaluation methods can be found in Fike (3).

3.10 CONCLUSION

In conclusion, it is only apt to mention the documentation of the basic library routines. The basic library is very dependent on the machine it was written for and on its environment as the routines are written in the Assembler code (section 3.2). It is important for the user to have access to the documents regarding the routines they so often use. In general, proper documentation is expected from those routines written for numerical algorithms. In recent years, more emphasis is being placed on documenting the all-important basic library; and performance testing and certification are carried out to certify the routines (10), (11). Dickinson (12) listed the following documentation format for mathematical routines. However, this can also be applied to the basic library sub-programmes.

The following document may be included with the basic library:

1. SCOPE:

   A short statement regarding what the programme does in non-mathematical terms is given.

2. MODEL:

The model which programmes use is stated generally in mathematical terms. Emphasis is made here to state what the routine does. The details of the algorithm are reserved in the appendix of the documentation.

3. LIMITATION:

Here all known limitations of the routines are listed. For example, the range of arguments and the range of the expected result.

4. ERROR ANALYSIS:

An error analysis of the probable errors that may occur in the routine. In particular, it should give detailed analysis of expected critical values (range of any) and the range of the errors.

5. LISTING OF PROGRAMMES:

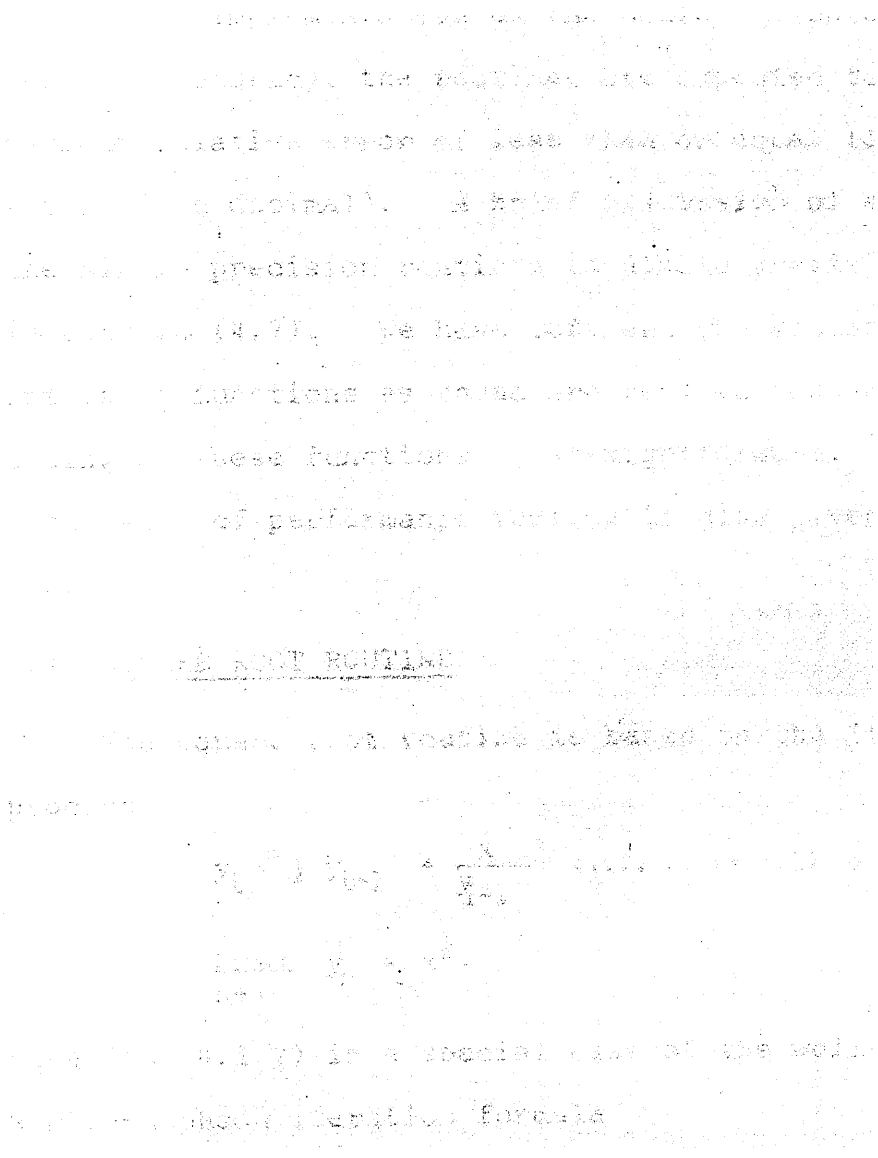
Listing of programmes is important for checking and for future extension to the (library) routines. Appropriate comments should be included in each step of the programme listing as it may be difficult to figure out the content if the programmes were written in the Assembler Code.

6. APPENDIX:

Here detailed description of the way in which the model is implemented is given. Outline of the procedure should be given if the algorithm is an implementation

of a literature procedure. Flowcharts should also be helpful in simplifying the description.

With proper documentation, the basic library can be frequently reviewed. Also this makes the library more portable. By 'portable' we mean small easily identifiable changes are necessary to transfer the software to a new environment. (Traub (13)).



CHAPTER FOUR

4.0 INTRODUCTION

In this chapter we shall discuss some of the routines in the basic library. Only single precision routines for square root, sine and cosine, logarithm, exponential, arctangent, and hyperbolic sine and cosine are considered. The purpose of the survey is to determine suitable algorithms for implementation on the MULTUM computer. In terms of accuracy, the routines are expected to give a maximum relative error of less than or equal to 2^{-23} (or 10^{-7} in decimal). A brief discussion of extending the single precision routines to double precision is given in section (4.7). We have left out the routines for intrinsic functions as these are very machine dependent. Coding of these functions is straightforward. A brief discussion of performance testing is also given in section 4.9.

4.1 SQUARE ROOT ROUTINE

The square root routine is based on the iterative process

$$y_i = \tfrac{1}{2}\, y_{i-1} + \frac{x}{y_{i-1}} \quad \ldots\ldots\ldots \text{(4.1.1)} \quad i = 1, 2, 3\ldots$$

$$\lim_{n\to\infty} y_n = x^{\frac{1}{2}}.$$

Equation (4.1.2) is a special case of the well-known Newton-Raphson iteration formula

$$y_i = y_i - \frac{f(y_i)}{f(y_i)} \quad \text{.....(4.1.2)} \quad i = 1, 2, \ldots$$

where $f(y) = y^2 - x$.

The sequence $y_1, y_2 \ldots$ defined by (4.1.1) converges quadratically to $\sqrt{x}$ (refer Fike (3) for proof). This implies that any (arbitrary) choice of the initial approximation y_0 will eventually give the square root of x (after a number of iterations). A reduction in the number of iterations needed to achieve a prescribed accuracy can be affected by expressing an arbitrary operand x in the form $x = m(x)2^{2n}$ where n is chosen so that $\frac{1}{2} \leq m(x) < 1$, then using $x^{\frac{1}{2}} = \sqrt{m(x)}2^n$. The reduction is simply a consequence of the fact that as the range of allowed m(x) values decreases, increasingly accurate initial approximations y_0 are possible. The number x is representable in the form

$$x = m(x)2^{e(x)} \quad \text{.....(4.1.3)}$$

In the floating-point representation equation (4.1.3) can be written as

$$x = \begin{cases} m(x)2^{2k} & \text{if } e(x) \text{ is even} \\ m(x).2.2^{2k} & \text{if } e(x) \text{ is odd} \end{cases}$$

and $\frac{1}{2} \leq m(x) < 1$ for positive value of x.

Negative values of x give imaginary roots and will not be considered here.

The square root is

$$x = \begin{cases} 2^k\sqrt{m(x)} & \text{if exponent } e(x) \text{ is even} \\ 2^k\sqrt{2m(x)} & \text{if exponent } e(x) \text{ is odd} \end{cases}$$

The computation of $\sqrt{x}$ is then reduced to find the square root of m(x) or 2m(x) in the range $[\frac{1}{2}\ 1)$. Write m(x) or 2m(x)

as m, the iterative process used will then be

$$y_i = \tfrac{1}{2}\left(y_{i-1} + \frac{m}{y_{i-1}}\right) \qquad i = 1, 2, \ldots$$

Let $E_i(m)$ be the relative error after i iterations.

Hence

$$E_i(m) = y_i\, m^{-\frac{1}{2}} - 1 \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (4.1.4)$$

satisfying

$$E_i(m) = \tfrac{1}{2} E_{i-1}^2(m)\,(1 - E_{i-1}(m))^{-1} \ldots\ldots\ldots\ldots\ldots (4.1.5)$$

These features, together with the economy of arithmetic operations involved in carrying out one cycle of the iterative process, imply that there is little to be gained by using sophisticated approximation to represent the initial approximation y_0. The choice of possible forms to represent x_0 is therefore limited to these three:-

a) linear approximation $\qquad y_0 = a_0 + a_1 m$

b) bilinear approximation $\qquad y_0 = b_0 + b_1 (m + b_2)^{-1}$

c) Quadratic approximation $\qquad y_0 = c_0 + c_1 m + c_2 m^2$

a) Linear Approximation

$$y_0 = a_0 + a_1 m \qquad \tfrac{1}{2} \leq m < 1$$

The constants a_0, a_1 can be obtained by applying Chekyshev's (minimax) Theorem (refer 3.5.2) and are found to be: $a_0 = 0.4173$, $a_1 = 0.5902$, (Eve, (19))

From equation (4.1.4)

$$E_0(m) = y_0\, m^{-\frac{1}{2}} - 1.$$

The maximum relative error in the initial approximation is

$$|E_0(m)| \leq 0.96 \times 2^{-7},$$

which can be deduced by application of Chekyshev's Theorem

and also taking the approximation $y_0 = 0.4173 + 0.5902m$ as the initial approximation.

From equation (4.1.5), we can see that the maximum relative error after two iterations, $E_2(m)$ is

$$0 \leq E_2(m) < 2^{-31}$$

and after three iterations

$$0 \leq E_3(m) < 2^{-63}$$

b) Bilinear Approximation

$$y_0 = b_0 + b_1 (m + b_2)^{-1}$$

Similarly, b_0, b_1, b_2 can be obtained by application of Chekyshev Theorem.

We have (Eve, (19))

$$y = 2.541639 - 4.837528/(m+2.137255) \quad \ldots\ldots\ldots\ldots (4.1.6)$$

From equation (4.1.4) and Chekyshev Theorem, together with the approximation (4.1.6) the maximum relative error $E_0(m)$ is

$$|E_0(m)| < 0.33 \times 2^{-10}$$

Repeated application of equation (4.1.5) gives

$$|E_0(m)| < \frac{1}{5} \times 2^{-47}$$

c) Quadratic Approximation

Similarly we have

$y_0 = 0.313553 + 0.890245m - 0.204445m^2$, $\frac{1}{2} \leq m < 1$ (refer Eve (19))

for which

$$|E_0(m)| < \frac{1}{3} \times 2^{-9}$$

and after one iteration, $|E_1(m)| < 2^{-22}$

and $|E_2(m)| < 2^{-44}$ after two iterations.

Comparison of the three approximations:

Table (1) gives the maximum relative errors corresponding to the three approximations and the number of iterations required.

TABLE (1): Comparison of Maximum Relative Errors $E_i(m)$

APPROXIMATION	$E_0(m)$	$E_1(m)$	$E_2(m)$
LINEAR	0.96×2^{-7}	2^{-15}	2^{-31}
BILINEAR	$.33 \times 2^{-10}$	2^{-21}	$\frac{1}{5} \times 2^{-47}$
QUADRATIC	$.33 \times 2^{-9}$	2^{-19}	2^{-44}

Comparison between the three types of approximations:

The linear approximation tends to converge more slowly than the other two cases. The maximum relative error registered in the bilinear and quadratic approximations is in the order of 2^{-45} as compared to that of the linear form, which is 2^{-31}. However, the initial approximation of y_0 in the bilinear and quadratic case requires more arithmetic operations. For example, in the bilinear approximation, two additions and one division are required to evaluate y_0 as compared to one addition and one multiplication in the linear form.

The floating-point representation in the MULTUM computer does not require accuracies in the order of 2^{-40}. The linear approximation is a better choice for implementa-

tion in the MULTUM computer. We can use fixed-point arithmetic to evaluate y_0 and the subsequent iterations. The approximation y after the second iteration will then be concatenated with the exponent, 2^k, to form the result of the square of x. If linear approximation were used, calculation of y_0 cannot be performed in fixed-point arithmetic as b_0, b_1 and b_2 are not pure fractions. Working on floating-point arithmetic, the accuracies in the order of 2^{-40} would serve no purpose at all since floating-point numbers require only 24 significant binary digits to represent the number.

4.1.7 ERROR PROPAGATION

Let argument have error ε and the exact value

$$x^* = x + \varepsilon$$

From binomial expansion

$$\sqrt{x(1+\varepsilon)} = \sqrt{x}\ (1+\tfrac{1}{2}\varepsilon - \tfrac{1}{8}\varepsilon^2 + \ldots\ )$$

The relative errors of the results are approximately one half of those of the arguments. Evaluations of y_i are performed in double-precision fixed-point arithmetic. In this case the full length of the accumulator is used. This will minimise the propagation error.

4.2 TRIGONOMETRIC FUNCTIONS

4.2.1 SINE AND COSINE SUB-ROUTINES

The relative accuracy of the computed value of a trigonometric function depends largely on the care exercised in the reduction stage. For this reason, it is

desirable to use an arithmetic higher than the working precision during this stage.

4.2.2 REDUCTION STAGE

Let the floating-point argument be x. Our task is to decompose the given argument x as

$|x| = (\frac{\Pi}{\alpha})n + r(\Pi/\alpha)$ where n is an integer and

α is a positive integer

and $0 \leq r < 1$

That is, given x, there is exactly one pair of values (n, r) that satisfies this relation. Then the reduced argument is either

$$g = (\frac{\Pi}{\alpha})r = |x| - n(\frac{\Pi}{\alpha}) \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (4.1.2)$$

or $$g' = (\frac{\Pi}{\alpha})(1-r) = (n+1)(\frac{\Pi}{\alpha}) - |x| \quad \ldots\ldots\ldots \quad (4.1.3)$$

depending upon the value of

$$m \equiv n \bmod 2\alpha \quad , \quad \alpha = 1, 2, 3\ldots$$

and the main computation will be sine or cosine of g or g'. From equation (4.1.2) and (4.1.3) loss of accuracy can occur in forming the difference between nearly equal quantities $|x|$ and an integer multiple of Π/α. For example, $|x|$ and $\frac{\Pi}{\alpha} n$ agree for the first k bits; then g may be in error in the last k bits leading to a large value of δg even though $\delta x \equiv 0$. The following example is due to Cody (8). If we let xxxxxxxx represent the computer bits devoted to normalised fraction part of a floating-point number x, a typical computation of g is as follows:

$$\begin{aligned} x &= \text{xxxxxxxx} \\ -n.\frac{\Pi}{\alpha} &= \underline{\text{xxxyyyyy}} \\ &\quad \text{000zzzzz} \end{aligned}$$

where the renormalisation of the intermediate result shifts zero bits into the low order positions. Suppose x is extended to double precision by the appendage of extra zero bits. Then computation of g is as follows:-

$$\begin{aligned} x &= .xxxxxxxx \quad 00000000 \\ -n\tfrac{\Pi}{\alpha} &= .xxxyyyyy \quad yyyyyyyy \\ \hline & .000zzzzz \quad zzzzzzzz \end{aligned}$$

normalised to give .zzzz zzzz.

The accuracy of g is then dependent on $n\frac{\Pi}{\alpha}$ alone. If we assume $\delta x \equiv 0$, we can see that

$$\delta \sin(g) = g \cot(g)\delta g$$

$$\delta \cos(g) = -g \tan(g)\delta g$$

Hence the magnitude of δg, the argument error (in the reduction stage) has a great influence on the accuracy of the computation over the reduced range.

In general, the constant α can be chosen arbitrarily. (Refer Hart (2).) For convenience, α is taken to be either 2 or 4 in our discussions. Consider the following algorithm for sine/cosine.

Define $z = \frac{2}{\Pi}|x|$ Let $z = n+r$

where n is an integer and r a fraction $0 \leq r < 1$

We then have

$$x = \frac{\Pi}{2} n + \frac{\Pi}{2} r$$

The full range of the variable x is reduced to $\left[0, \frac{\Pi}{2}\right]$ in our case here. If we restrict r in the range $\left[-\frac{1}{2}, \frac{1}{2}\right]$ the full range of x will then be reduced to $\left[-\frac{\Pi}{4}, \frac{\Pi}{4}\right]$. We shall only consider the first case.

The following trigonometric identities enable us to

cover the negative values of x by adding integer constants to n in the reduction stage.

$$\text{Cos}(\pm x) = \text{Sin}(\tfrac{\Pi}{2}+x)$$

$$\text{Sin}(-x) = \text{Sin}(\Pi+x)$$

If sine of a negative argument is desired, add 2 to n. If cosine is desired, add 1 to n. This adjustment of n reduces the general case to the computation of sin(x) for $x \geq 0$.

Let me = n mod 2α

if $\alpha = 2$, $\quad m = n \bmod 4$.

Using the identity sin (A+B) = sin(A)Cos(B) + sin(B)cos(A) we have, for $m = 0$, $\sin(x) = \sin(\frac{\Pi}{2}r)$

$m = 1$, $\sin(x) = \sin(\frac{\Pi}{2}+\frac{\Pi}{2}r) = \cos(\frac{\Pi}{2}r)$

$m = 2$, $\sin(x) = -\sin(\frac{\Pi}{2}r)$

$m = 3$, $\sin(x) = \sin(\frac{3\Pi}{2}+\frac{\Pi}{2}r) = -\cos(\frac{\Pi}{2}r)$

The formulae reduce each case to the computation of either $\sin(\frac{\Pi}{2}r)$ or $\cos(\frac{\Pi}{2}r)$ where $0 \leq r < 1$.

If $\alpha = 4$, the reduction stage will be

$$|x| = \tfrac{\Pi}{4}n + \tfrac{\Pi}{4}r \ , \quad 0 \leq r < 1, \text{ n is an integer.}$$

The full range is then reduced to $\left[0\ \frac{\Pi}{4}\right]$.

Now m = n mod 8, and we have

$m = 0$, $\sin(x) = \sin(\frac{\Pi}{4}r)$

$m = 1$, $\sin(x) = \cos\{\frac{\Pi}{4}(1-r)\}$

$m = 2$, $\sin(x) = \cos(\frac{\Pi}{4}r)$

$m = 3$, $\sin(x) = \sin\{\frac{\Pi}{4}(1-r)\}$

$$m = 4, \quad \sin(x) = -\sin(\tfrac{\Pi}{4}r)$$

$$m = 5, \quad \sin(x) = -\cos\tfrac{\Pi}{4}(1-r)$$

$$m = 6, \quad \sin(x) = -\cos(\tfrac{\Pi}{4}r)$$

$$m = 7, \quad \sin(x) = -\sin\{\tfrac{\Pi}{4}(1-r)\}$$

The computation is also reduced to the evaluation of either $\sin(\frac{\Pi}{4}r_1)$ or $\cos(\frac{\Pi}{4}r_1)$ where r_1 is either r or (1-r) and is within the range $0 \leq r_1 < 1$.

4.2.3 CHOICE OF α

From Chapter Three, we infer that the smaller the approximation range the less terms a polynomial or rational approximation must contain to approximate a function with a specified accuracy.

Suppose the sine function is approximated by a polynomial, say $\sin(\Pi/\alpha x) \approx xP(x^2)$.
The following table (2) is excerpted from Hart (2):-

TABLE (2) Comparison of Precision for Different α and n

Maximum Relative Error = 2^{-x}

α/n	3	4	5
α=2	x=20.03	x=27.47	x=36.14
4	28.20	37.67	47.66
6	32.92	43.55	54.71

From Table (2) it can be seen that precision increases when x increases. For example, when $\alpha \approx 4$, the maximum relative error is less than or equal to $2^{-28.20}$, if a poly-

nomial of degree 3 in x^2 is used to approximate $\sin(\frac{\Pi}{\alpha}x)$. When $\alpha=2$, maximum relative error is $2^{-20.03}$ for a polynomial of the same degree. For practical consideration, $\alpha=4$ is a better choice than $\alpha=2$, since floating-point number representable by the MULTUM computer (refer section 2.3.1) has a significant of 24 binary digits. Hence the computation for either sine or cosine is performed using the Chebyshev interpolation of degree 3 in r^2 (refer section 3.8.1). The maximum relative error in the sine approximation is $2^{-28.1}$ and that of cosine is $2^{-24.6}$.

We have

$$\mathrm{Sin}\tfrac{1}{4}\Pi x \approx r\,P(r^2) = r\sum_{r=0}^{3} a_n (r^2)^n$$

The coefficients are due to Hart (2) (Table sin 3040).

$$a_0 = +\ 0.78539\ \ 816 \times 10^{0}$$
$$a_1 = -\ 0.80745\ \ 433 \times 10^{-1}$$
$$a_2 = +\ 0.24900\ \ 010 \times 10^{-2}$$
$$a_3 = -\ 0.35950\ \ 439 \times 10^{-3}$$

and for $\mathrm{Cos}\tfrac{1}{4}\Pi r \approx P(r^2) = \sum_{n=0}^{n=3} b_n (r^2)^n$

$$b_0 = +.99999\ 997 \times 10^{0}$$
$$b_1 = -.30842\ 417 \times 10^{0}$$
$$b_2 = +.15849\ 684 \times 10^{-1}$$
$$b_3 = -.31872\ 780 \times 10^{-3}$$

4.3 LOGARITHM (ALOG, ALOG10)

The natural logarithm, ln(x), (or written as $\log_e(x)$) is the inverse of the exponential of x.

i.e. $\ln \exp(x) = x = \exp \ln(x)$

In addition, logarithm is defined only for real positive values of x.

RANGE REDUCTION

It would be most impractical and cumbersome to find an approximation for ln(x) in the range $0 < x < \infty$ or in the range of x representable by the machine. Thus it is necessary to reduce x to an interval that will give a rapid convergent for the approximation.

A typical range reduction will be the decomposition of the argument x as the product

$$x = 2^n.m \quad \text{where} \quad 0.5 \leq n < 1 \qquad (4.3.1)$$

Then the logarithm of x will be

$$\ln(x) = n \ln(2) + \ln(m) \qquad (4.3.2)$$

The approximation for ln(x) is then reduced to finding a suitable approximation for ln(m) in the interval

$$0.5 \leq m < 1$$

This reduction scheme is easily realised as the standardised floating-point number is of the form given in equation (4.3.1).

For practical and accuracy requirements, the interval $\frac{1}{2} \leq m < 1$ can be subdivided into 2^k intervals, $[\,2^{-j/2k},\ 2^{(1-j)/2k}]$ by the substitution

$$g = 2^{(1-j)/2t}_{m}$$

then

$$\ln(x) = \ln(2^n.2^{j-1/2k})$$

$$= n \ln(2) + \frac{j-1}{2^k} \ln(2) + \ln(g)$$

It only remains to find an approximation for ln(g) that

will give a rapid convergence in the chosen interval.

Two forms of approximation are considered for ln(m). The first is a simple polynomial or rational approximation in m.

$$\ln(m) = R(m) \qquad \text{.....(4.3.3)}$$

The second is of the form

$$\ln(g) = \left(\frac{g-1}{g+1}\right) R \left(\frac{g-1}{g+1}\right)^2 \qquad \text{..... (4.3.4)}$$

Table (3) gives an indication of how rapidly these approximations converge.

TABLE 3 Comparison between Approximation of Form (i) $\ln(m) \simeq R(m)$, (ii) $\ln(g) = zR(z^2)$ where $g = (g-1)/(g+1)$ maximum relative error = 2^{-x}

	INTERVAL	x	DEGREE n_1	n_2	FORM OF APPROXIMATION
(i)	$[\frac{1}{2}, 1]$	25.01	8		$\ln(x) \simeq P(x)$
	$[\frac{1}{2}, 1]$	7.05	3	2	$\ln(x) \simeq P(x)/Q(x)$
(ii)	$[\frac{1}{\sqrt{2}}, \sqrt{2}]$	25.51	2		$\ln(x) \simeq z\ P(z^2)$
	$[\frac{1}{\sqrt{2}}, \sqrt{2}]$	28.16	1	1	$\ln(x) \simeq z\ P(z^2)/Q(z^2)$

The values in Table (3) are excerpted from Hart (2).

Approximation in the form $\ln(x) = zP(z^2)$ where $z = (x-1)/(x+1)$ gives a fast convergence at the expense of two extra additions and one division and one multiplication to form z^2 in addition to the arithmetic for evaluating the polynomial.

Subdivision of the interval $[\frac{1}{2}, 1]$ into 2^k intervals may require k comparisons to determine which interval contains m and also extra storage is required to store the quantities $(j-1)/2^k$. The trade-off between storage space, accuracy and speed will deter the possibility of having too many subdivisions. As an alternative to approximate function ln(m) in $[\frac{1}{2}, 1]$ we may use the substitution

$$s = am \quad \text{where} \quad \tfrac{1}{2} \leq m < 1$$

and a is a constant,

in which case s is in the interval $[\frac{a}{2}, a]$

where $s \in [\frac{1}{\sqrt{2}}, \sqrt{2}]$ and

$$\ln(x) = 2^n \frac{1}{\sqrt{2}} s$$

$$= (n-\tfrac{1}{2}) \ln(2) + \ln(s)$$

To calculate the logarithm function ln(x), the Chebyshev polynomial expansion on $[\frac{1}{2}, 1]$ is truncated and transformed into a power series.

$$\ln(x) = -\tfrac{1}{2}\ln 2 - \sum_{k=\phi}^{\infty} \frac{4p^{2k+1}}{2k+1} T_{2k+1}\left((\sqrt{2}+1)^2 \frac{1-\sqrt{2}x}{1+\sqrt{2}x}\right)$$

$$\text{where } p = \frac{4\sqrt{2}-1}{4\sqrt{2}+1}$$

This expansion is obtained by using the orthogonal properties of the polynomial, $T_n(x)$ to determine the constants in the expansion,

$$\ln(x) = \tfrac{1}{2}a_0 + \sum_{k=1}^{\infty} a_k T_k(x) \quad \text{by integration.}$$

The economised Chebyshev polynomial expansion, is

$$\ln(x) = \tfrac{1}{2}\ln 2 + \sum_{k=\phi}^{3} a_{2k+1} u^{2k+1}$$

where $u = \dfrac{x - \frac{\sqrt{2}}{2}}{x + \frac{\sqrt{2}}{2}}$, $\frac{1}{2} \leq x \leq 1$

A maximum error of 2^{-32} is incurred if the approximation is used. The constants are

a_1 = 1.999 999 993 788

a_3 = 0.666 669 470 507

a_5 = 0.399 659 100 019

a_7 = 0.300 974 506 336 (refer (18))

Evaluation of the parameter u will cancel a number of significant digits if x is very nearly $\frac{1}{\sqrt{2}}$. Extended precision arithmetic is required to determine u accurately as the whole accuracy of the routine depends on this.

Alternatively, we can have the following algorithm. Suppose that we have a polynomial approximation $\ln(m) = zP(z^2)$ where $z = \frac{m-1}{m+1}$, $\frac{1}{\sqrt{2}} \leq m < \sqrt{2}$.

To calculate ln(x), we write

$$x = \alpha^n m \qquad \tfrac{1}{2} \leq m < 1$$

and $\ln(x) = n \ln(2) + \ln(m)$.

If $\frac{1}{\sqrt{2}} \leq m < 1$, evaluation of ln(x) is obvious.

However, for $\frac{1}{2} \leq m < \frac{1}{\sqrt{2}}$, we need to have an approximation, $zP(z^2)$ in this range or use a substitution in order to make use of the same polynomial approximation for the range $\left[\frac{1}{\sqrt{2}}, \sqrt{2}\right]$.

Let the substitution be

$$s = 2m,$$

this transforms $s \in \left[1, \sqrt{2}\right] \subset \left[\frac{1}{\sqrt{2}}, \sqrt{2}\right]$

and evaluation of ln(x) will be

$$\ln(x) = (n-1)\ln(2) + \ln(s).$$

Then

$$\ln(s) \simeq zP(z^2) \text{ where } z = \frac{2m-1}{2m+1} \equiv \frac{m-\frac{1}{2}}{m+\frac{1}{2}}.$$

An example for the polynomial approximation $zP(z^2)$ is

$$zP(z^2) \simeq \sum_{n=0}^{2} a_{2n+1} z^{2n+1}$$

where $a_1 = 2.0000008$

$a_3 = 0.66644078$

$a_5 = 0.41517739$ (refer Kuki (7))

A maximum relative error for this approximation is 10^{-7},

(or A maximum relative error for this approximation is 10^{-7}

(or $2^{-23.2}$)

The evaluation of logarithm to base 10 is obtained from the relation

$$\log_{10}(x) = \log_2(E)\ln(x)$$

where $\log_2(E) = 1.4426\ 9504$

4.4 EXPONENTIAL FUNCTION (EXP(x))

The range of argument x, for the exponential function exp(x) (or written e^x), runs from $-\infty$ to $+\infty$. Range reduction is practical and necessary for exponential routines.

a) RANGE REDUCTION

We may write

$$\exp(x) = 2^{x/\ln(2)} = 2^{n+m} = 2^n 2^m \quad \ldots\ldots\ldots\ldots (4.4.1)$$

with n integer and $0 \leq m < 1$.

This reduces to finding an approximation for 2^m as the factor

2^n may be taken into account by adding n to the exponent in floating-point.

Now

$$2^m = e^{\ln(2)^m}$$

$$= e^{m \ln(2)}$$

$$= (e^{\frac{1}{2}m \ln(2)})^2$$

if we define $y = m \ln(2)/2$

Then

$$y \in \left[-\frac{\ln 2}{2}, \frac{\ln 2}{2}\right]$$

On this interval, we can find a polynomial or rational approximation to e^y.

For example

$$(e^y) = 1 + \frac{2y}{a_0 - y - \frac{a_1}{b_1 + y^2}}$$

is nearly the best (in Chekyshev's sense) rational approximation which has a relative error of less than 10^{-9} (or $2^{-29.6}$)

The constants are

$$a_0 = 12.015\ 167\ 538\ 7500$$

$$a_1 = -601.804\ 266\ 697\ 9565$$

$$b_1 = 60.090\ 190\ 731\ 9260 \qquad \text{(refer (18))}$$

Also we can express

$$\exp(x) = 2^{x \log_2 e}$$

$$= 2^{n-m}$$

$$= 2^n . \bar{2}^m \quad \text{where } 0 < m < 1 \ldots\ldots\ldots\ldots (4.4.2)$$

Evaluation of exponential of x (exp(x)) is therefore confined to finding a suitable approximation for 2^m (or 2^{-m}) instead of finding an approximation for e^y in the earlier

case. If a polynomial approximation, P(x) is used to approximate 2^m i.e. $2^m \simeq P(m) \qquad |m| < 1$ then value of the reduced argument from equation (4.4.2) is always negative by a proper choice of n.

Let $P(m) \simeq \sum_{i=0}^{5} a_i m^i$

where
$a_0 = 0.999\ 999\ 93$
$a_1 = 0.693\ 142\ 26$
$a_2 = 0.240\ 172\ 24$
$a_3 = 0.552\ 798\ 30 \quad 10^{-1}$
$a_4 = 0.918\ 869\ 80 \quad 10^{-2}$
$a_5 = 0.938\ 811\ 00 \quad 10^{-3}$ (refer (18))

The maximum relative error is less than 10^{-7} (or $2^{-23.1}$)

As accuracy of 10^{-7} is only required by the computer, the approximation of exp(x) by $P(m) \simeq 2^m$ seems a better choice.

The routine for exponential, exp(x) should provide alarms in case exp(x) exceeds machine capacity. This is most easily determined by checking that n in equation (4.1.2) is not larger than the largest allowable exponent. Also provision should be made to return zero if n is less than the smallest allowable exponent. In cases where x is very near zero, 1 is returned as the answer.

4.5 INVERSE TANGENT (ATAN, ATAN2)

The value of inverse tangent function is the angle at which the corresponding tangent attains a special value. The tangent function is periodic and takes on the same

interval twice in each interval of length 2Π, the inverse tangent is multiple valued. Using the following identities

$$\text{arc tan } (-x) = -\text{arc tan } (x)$$

and

$$\text{arc tan } \left(\frac{1}{|x|}\right) = \frac{\Pi}{2} - \text{arc tan } (x)$$

the range $(-\infty, \infty)$ can be reduced to $[0, 1]$

Thus arc tan (x) can be expressed in terms of arc tan (y) where $y = g(x)$ and $g(x) = x$ or $\frac{1}{x}$.

Further reduction of the interval $0 \leq y \leq 1$, (or $0 \leq y \leq \tan\frac{\Pi}{4}$) to a smaller interval gives a better polynomial approximation in the smaller range than the larger range $[0, 1]$

Table (4) illustrates the influence of the interval on the accuracy and speed of the polynomial approximation. The values in Table (4) are excerpted from Hart (2).

TABLE (4)

Arc tan $\simeq xP(x^2)$

Comparison between Different Ranges,

Maximum Relative Error = 2^{-x}

RANGE	x	DEGREE OF POLYNOMIAL
$\left[0, \tan \frac{\Pi}{32}\right]$	27.90	2
	36.93	3
$\left[0, \tan \frac{\Pi}{16}\right]$	21.85	2
	28.90	3
$\left[0, \tan \frac{\Pi}{12}\right]$	25.54	3
	31.69	4
$\left[0, \tan \frac{\Pi}{8}\right]$	15.74	2
	20.71	3
	25.71	4
$\left[0, \tan \frac{\Pi}{4}\right]$	20.56	6
	23. 6	7
	25.97	8

From Table (4) we can see that a polynomial of degree 7 in x^2 is required to approximate arc tan (x) in the range $0 \leq x \leq 1$. To achieve the same accuracy, that is a maximum relative error of 2^{-25}, a polynomial of degree 3 in x^2 is required if the argument x is in the range $\left[0, \tan \frac{\Pi}{12}\right]$.

Hence the following algorithm is used.

When $x < 2-\sqrt{3}$ then $z = x$ and $c = 0$

and $x \geq 2-\sqrt{3}$ then $z = \frac{x\sqrt{3}-1}{x+\sqrt{3}}$ and $c = \Pi/b$

and $\text{arc tan}(z) = \sum_{k=0}^{3} a_{2k+1} z^{2k+1}$

$\text{arc tan}(y) = \text{arc tan}(z) + c$

The coefficients a's are

$a_1 = +.99999\ 99797\ 73$

$a_3 = -.33332\ 42344\ 5$

$a_5 = +.19935\ 72694$

$a_7 = -.12813\ 3334$ (refer Hart (2))

For $\text{arc tan}(\frac{x_1}{x_2})$, the algorithm is the same as that for one argument.

If $x_2 = 0$

$$\text{arc tan}(\frac{x_1}{x_2}) = \text{sign}(x_1)\ \frac{\Pi}{2}$$

if $|\frac{x_1}{x_2}| > 2^{24}$, the value $x = |\frac{x_1}{x_2}|$ is an integer as the mantissa of the floating-point number is 24 digits and the value $\frac{1}{x}$ is very small and $\text{arc tan}(\frac{1}{x}) = 0$

if $|\frac{x_1}{x_2}| > 2^{24}$, $\text{arc tan}(\frac{x_1}{x_2}) = (\text{sign } x_1).\frac{\Pi}{2}$

otherwise if $x_2 > 0$ the answer $= \text{arc tan}(\frac{x_1}{x_2})$

and if $x_2 < 0$ the answer $= \text{arc tan}(\frac{x_1}{x_2}) + (\text{sign } x_1)\Pi$.

Care should be taken in coding the algorithm. Instead of computing the value $\sqrt{3}x-1$ directly, $(\sqrt{3}-1)x-1+x$ is computed to avoid loss of significant digits.

4.6 HYPERBOLIC SINE AND COSINE (SINH and COSH)

The hyperbolic sine and cosine functions are defined by equations (4.6.1) and (4.6.2)

$$\sinh(x) = (e^{x}-e^{-x})/2 \quad \text{.....(4.6.1)}$$

$$\cosh(x) = (e^{x}+e^{-x})/2 \quad \text{.....(4.6.2)}$$

The value of sin h (x) is symmetrical about the y-axis and for all values of x, sin h (x) $\geq$ 1. By symmetry, we need only consider the argument range $0, \infty$. To get a better result in sin h (x) near the origin, that is for argument in the range $[0, 1]$, we may compute sinh (x) by the polynomial approximation

$$\sin h\ (x) \approx xP(x^2)$$

where

$$xP(x^2) \approx \sum_{i=0}^{2} a_{2i+1}\, x^{2i+1}$$

and

$$a_1 = 1.0000\ 00000\ 1327$$

$$a_3 = 1.6666\ 65805\ 763$$

$$a_5 = .83416\ 01527 \times 10^{-2}$$

(refer Hart (2), Table SINH 1962)

The maximum relative error of this approximation is less than $2^{32.4}$

If $0 \leq x \leq \frac{1}{2}$ were chosen, the best (in Chekyshev's sense) polynomial approximation for sinh (x) is given by

$$xP(x^2) = \sum_{i=0}^{2} b_{2i+1}\, x^{2i+1}$$

and

$$b_1 = 1.0000\ 00095\ 55$$

$$b_3 = .16666\ 97150\ 46$$

$$b_5 = .8077\ 9341 \times 10^{-2}$$

(Hart (2) Table SINH 1982)

The maximum relative error in this case is less than $2^{-23.4}$. An accuracy in the order of 2^{-30} is not required. It is the best choice in approximating the sinh(x) in the range $[0, \frac{1}{2}]$. For values of $|x|>\frac{1}{2}$, sinh(x) is computed as

$$\sinh(x) = (\text{sign } x)\ \frac{w-w^{-1}}{2}$$

where $$w = e^{|x|}$$

From equation (4.6.2), Cosh(x) behaves like the exponential function. To obtain the value for hyperbolic cosine of x, Cosh(x) is computed as

$$\text{Cosh}(x) = (w+w^{-1})/2$$

where $w = e^{|x|}$

The real exponential sub-programme is used to compute the value of w in both cases.

4.7 DOUBLE PRECISION BASIC LIBRARY ROUTINES

So far, we have only discussed single precision routines for the basic library. Double precision routines are required to provide higher level language support for double precision (or extended precision) floating-point arithmetic. In the MULTUM computer, no double precision floating-point instructions are available. In addition to the explicitly and implicitly double precision mathematical routines, double precision arithmetic simulators and input/output conversion programmes should be included in the library.

Double precision arithmetic simulation is required to provide the complete set of double precision instructions

that is not available in the computer (see section 2.4).

A routine is required for base conversion of an input decimal number into an internally useable form, including the conversion of up to 15 decimal digits of input into the double precision binary form. Another routine is also required to handle output conversion including conversion of a double precision number to a decimal number of up to 15 decimal digits. This routine handles the conversion and format of the print field.

Double precision mathematical functions can be made available using the same algorithms given earlier except the approximation used. For example in the sine/cosine routine, to extend the routine to a double precision routine, we approximate sine/cosine function by Chebyshev interpolation of degree 6 instead of the Chebyshev interpolation of degree 3 in r^2 (see section 4.2). Computations are done in double precision arithmetic. Coefficients of the polynomial approximation are given to the double precision accuracy. However, difficulties do arise in the reduction stage where we need extra accuracy to calculate the reduced argument. (See section 4.2.2). Similarly, we can extend the logarithm, exponential, inverse trigonometric functions by finding the appropriate 'double precision' approximation. In the square root routine we can either iterate a few more terms using the same starting approximation (but work in double precision arithmetic) or use another starting approximation given in section (4.1).

More sophisticated methods of designing a double precision routine can be done using a Chebyshev best fit approximation by rational functions. The approximation can either be presented by a Thiele-Tyre fraction or a Jacobi-fraction. If Thiele-Tyre fraction were used, P. Spelluci (18) found that the evaluation would be slow but well behaved with respect to error propagation. If the approximations were presented by the Jacobi fractions, evaluation was fast but needed provision for guard digits to preserve the full precision of approximation.

Considering the speed and accuracy required in the MULTUM computer, the method of extending the available logarithm is a better method.

4.8 SELF-CONTAINED POWER ROUTINES

We have left out one important routine in our discussion of library functions, viz. the exponentiation or power routines. The exponentiation routines are some of the most important in the library corresponding to the Fortran operation "**" operator. The survey of floating-point power routines reported in Clark and Cody's paper (11) shows that error in the last 7 to 10 digits is common even for moderate arguments.

Normally, the power routines are considered as management routines (see section 3.1.3). The exponential and the logarithm routines were used to compute the value x**y. The standard approach to floating-point exponenti-

ation involves the computation

$$x**y = \exp y \ln (x)$$

Let $w = y \ln (x)$

and $z = \exp (w)$

then $\delta z \simeq \exp (w)\, \delta w$

i.e. $\delta z \simeq \Delta w$

If w is computed in working precision, the word length of the computer assures us that Δw, hence δz, is likely to be large whenever w is large. This phenomenon is independent of the two primary routines (viz. logarithm and exponential). To avoid this error, both ln (x) and w should be computed to higher than working precision, assuming that arguments x and y are precise. If the required reduction

$$w = n \ln (2) + f, \qquad |f| \leq \ln (2)/2$$

is performed in the higher precision, the final computation will be essentially as accurate as the exponential computation.

If the working precision is single precision, the extra precision required for this approach can be obtained by doing the steps in either fixed-point arithmetic or in double precision floating-point if that is available.

The raising of the status of the power routines to that of primary routines will improve the accuracy of the routine. This can be done by computing the logarithm and exponential routines in the power routine itself. Such self-contained power (exponentiation) routines were

suggested by Cody, Clark and Kuki (15), (16). Working versions of self-contained single precision routines of this type have been available on the IBM 7094 at the University of Toronto and on the CDC 3600 at Argonne National Laboratory since early 1960's, and on IBM 360 at Argonne and IBM 7094 at the University of Chicago since about 1967 (16).

The elevation from management status to primary status increases the accuracy and speed of the routine but has a penalty on the storage requirements. Some of the overall storage can be retrieved by reducing the standard exponential and logarithm routines to appropriate entries in the corresponding self-contained power routines. Unfortunately, the execution time of the logarithm routine and the exponential routine is considerably slower. Clark (15) gives a 50% increase in execution time for the logarithm and about 10% increase in time for the exponential routine.

4.9 PERFORMANCE TESTING OF BASIC LIBRARY SUB-ROUTINES

The certification of basic library sub-routines for computers is relatively easy compared with the general numerical sub-routines. In any case, it is often a difficult job to define a concrete measure of performance for a particular type of routine as the details of the performance testing vary from one type of sub-routine to another.

Distribution is required between actual testing of performance of a sub-routine and judgement of quality based

upon the results of the testing.

Rigorous testing is required to ensure that statistics and facts from the testing are indisputable. Quality testing should attempt to determine error as precisely as possible, in order to be meaningful to any potential user.

To begin with, we shall look into the question of errors made by sub-routines, as quality testing will largely depend on this. Clark and Cody (11) classified these errors into three types, namely (i) transmitted error, (ii) analytic truncation error and (iii) analytic rounding error. Analytic truncation error is error made in the finite approximation to the infinite process. Analytic rounding error is error made in the computation of the approximation. Together, these two types of error are termed as generated error. (Refer section 3.4.2). Transmitted error is error due to the arguments. (Refer section 3.4.)

4.9.1 ERROR TESTING

The simplest method of testing is direct testing of computed function values against published tables. However, this method is most unsatisfactory. Comparison values of this type will involve human handling of the standard values. In practical terms, this is inefficient and likely human errors will be involved in transcribing the tabular values. Entries in published tables are normally very sparse. Values returned by a routine may agree with tabulated values and may yet give poor results

for other arguments. Also comparison against tables always involves conversion of tabulated decimal arguments into binary arguments. This leads to generated errors from conversion routines. In addition, transmitted error and the subsequent error in conversion of computed results back to decimal form will contaminate the final error statistics. In some sub-routines generated error is difficult to detect unless it is very large.

All these point to the need for automation in quality testing. Machine generated arguments and standards should be used to reduce the drawbacks stated. Clearly, the best tests involve a large number of arguments that are dense and not restricted to relatively small finite sets of 'nice' arguments given in most tables.

4.9.2 BIT PATTERN COMPARISON

Standard arguments generated by the computer will need greater precision than that of the value under test. For example, if a single precision routine were under test, we would then require to compute the same function more accurately than single precision (say, double precision).

Testing arguments are generated by the computer. These pseudo-random numbers are either uniformly distributed or exponentially distributed depending on what is required. The arguments are computed in test precision and extended to higher precision by appending appropriate low order zeros. Computation can then be carried out with 'identical' arguments in both single and double precision routines.

The possibility of transmitted error is eliminated by this process. Rounding the double precision result is normally preferred to truncating to single precision (10). A bit-pattern comparison of the results is obtained using fixed-point subtraction. Tables of the frequency of the difference in the bit pattern between the rounded single precision result (from double precision) are made. These statistics give an indication as to how well the sub-routine produces the machine number closest to the correct function value.

Additional statistics may be obtained from the above procedure. The maximum relative error and the root mean square relative error can be easily obtained. However, computation of the maximum relative error and the root mean square relative error should be in higher precision. Extensive testings were performed by Clark and Cody (11) using the unrounded value of the standard value to obtain the root mean square relative error and the maximum relative error.

The choice of test intervals and the distribution of random arguments is greatly related to the internal structure of the sub-routine under test. Special tests are required for critical ranges such as neighbourhood points where intermediate underflow or overflow may occur. Also, tests for the error return by using arguments at and just beyond the limits of acceptability are required. Extremely large and small numbers are tested to check for overflow and

underflow problems.

4.9.3 TIMING

Time checks are normally obtained by performing the sub-routine for several thousand random arguments using a loop of some sort. The overhead for the loop can be obtained by testing an identical loop with the test sub-routine replaced by a special sub-routine whose only executable instruction is a return to a calling programme. Instead of testing with several thousand random arguments, this can be replaced by a fixed argument and then perform the routine several thousand times. It is important to have a sufficient number of time round the loop to minimise the effect of the coarseness of the clock.

For double precision sub-routines a computer with larger word length may be used to generate the standard values. Pseudo-random double precision floating-point numbers in the larger computer word machine are generated and then converted to the length of the machine under test by rounding and zeroing out the extra bits. These converted arguments are then used to generate the corresponding function values on the 'larger' machine and the results are rounded to the 'test' computer number format. The pairing of arguments and function values may be transmitted to the 'test' computer via magnetic tape and the testing carried out on these, in a manner analogous to the single precision testing. The root mean square relative error and maximum relative error computed is not

as accurate as those for the single precision routine if higher precision arithmetics are not available. Such procedures were, in fact, used by Clark and Cody in finding the performance statistics of the Fortran IV (H) Library for IBM sys./360 (10). The large machine used was the CDC 3600 where the double precision format is an 80-bit mantissa.

4.10 CONCLUSION

The need for double precision floating-point arithmetic by hardware is obvious if we are to design accurate and efficiént basic library routines. Double precision arithmetic is required in some stages of the evaluation in order to obtain a last-digit accuracy for the result. For example in the argument reduction stage in most of the routines, the accuracy of the routines depends on the last-digit accuracy of the reduced argument. Computation of the reduced argument at this stage by double precision arithmetic will ensure this. Unfortunately, hardware instructions for double precision arithmetic are not available in the MULTUM and this greatly affects the design of the library. Simulated double precision (floating-point, refer section 2.4) arithmetic by software has heavy penalty in the time of execution of the routine. Also, double precision arithmetic is required in testing and certification of the library. Implementation of the double precision basic library routines without double precision floating-point arithmetic by hardware is most inefficient and very slow.

Also double precision arithmetic is needed in the generation of accurate constants to assist in the preparation of full accuracy single precision routines. In many cases, these will be the coefficients in expansions of functions, in particular, as power series, series of Chebyshev polynomials and continued fractions. Accurate zeros for routines are needed to preserve relative accuracy in constructing function routines. Having considered these factors, it is strongly urged that double precision floating-point arithmetic be included as hardware operations by the computer manufacturers.

The routines have not been coded. Careful consideration should be given in the actual coding as errors due to straightforward coding can be substantial. For example, in the sine/cosine routine, using straightforward coding to obtain $(2/\Pi)x-n$ (refer section 4.2) introduces an absolute error approximately equal to $n2^{-p}$ where p is the binary precision of the machine, since $Sin(\frac{\Pi}{2} r) \simeq 2r$, an absolute error of $n.2^{-p+1}$ in the answer from this source. This is the same as the effect of the minimal round-off error in the argument. Since Sin(x) becomes 0 periodically, this means that generated relative error will become infinite periodically. It can be concluded that straightforward (coding) reduction generates an error approximately equal in magnitude to the effect of the minimal round-off error in the argument.

The trade off between cost and refinement is a matter of individual judgement. Kuki (7) suggested that an increase in execution time by 10% and 15% in storage to attain a virtual last-digit accuracy is acceptable.

Routines like tanh(x), arc sin(x) and arc cos(x), which are not in the ASA Fortran standard,should be included in the library routines. If these are included they should be written as primary routines and not as secondary routines. That is, computation of arc sin(x) and arc-cos(x) (for example) will not call upon the arc tan(x) routine to perform the main calculation. The elevation of status from secondary to primary often increases the efficiency and accuracy of the routines.

BIBLIOGRAPHY

1	Kuki, H. and Ascoly, J.	Fortran Extended Precision Library, IBM SYSTEM JOURNAL, No. 1, 1971, pg. 39-61
2	Hart, Joh. F.	Computer Approximations - (1968), John Wiley
3	Fike, C.T.	Computer Evaluation of Mathematical Function (1968) Prentice-Hall
4	Snyder, M.A.	Chebyshev Methods in Numerical Approximation (1966) Prentice-Hall
5	Rice J.R.	The Approximation of Function. Volume I. 1964, Addison-Wesley.
6	Achieser, N.I.	Theory of Approximation. Ungar, New York. English translation by C.J. Hyman (1956)
7	Kuki, H.	Mathematical Function Sub-programmes. Basic System Libraries - Objectives, Constraints and Trade-off. Mathematical Software (Ed.Rice) 1971. Academic Press
8	Cody, W.J.	Software for the Elementary Functions - Mathematical Software (Ed. Rice) 1971. Academic Press.
9	Veidinger, L.	On the Numerical Determination of the Best Approximations in the Chebyshev Sense. Numer.Math. 2, 99-105.
10	Cody, W.J.	Performance Testing of Function Sub-routines, AFIPS Conf. Proc. 34, pg. 759-763.
11	Clark, N.A. and Cody, W.J.	Performance Statistics of the

		Fortran IV (H) Library for IBM SYSTEM/360. Argonne National Laboratory May 1967; ANL-7321
12	Dickinson, A.W. Herbert, V.P.	The Development and Maintenance of a Technical Sub-programme Library. Mathematical Software (Ed. Rice) 1971 - Academic Press
13	Traub, J.F.	High Quality Portable Numerical Mathematics. Mathematical Software (Ed. Rice) 1971. Academic Press
14	Cody, W.J.	The Influence of Machine Design on Numerical Algorithms. Vol.30 AFIPS Conf. Proc. 1967, pp.305-309
15	Clark, N.W. and Cody, W.J.	Self-contained Exponentiation. AFIPS Conf. Proc. Vol.35, (1969) pg.701-706
16	Clark, N.W., Cody, W.J., Kuki, H.	Self-contained Power Routines. Mathematical Software (Ed. Rice) 1971, Academic Press
17	P. Spelluci	Double Precision Approximations to the Elementary Functions Using Jacobi-Fractions. Numer. Math. 18. (1971) pg.127-143
18	POP-11	Paper Tape Software Programmeing Handbook Dec.-11- GGPB-D
19	Eve, J.	Starting Approximations for the Iterative Calculation of Square Roots (1963). Comput. J. 6, 274-276.

APPENDIX 1

In this appendix, a set of Usercode MULTUM Usercode Language procedures for single precision floating-point arithmetic is given. These are grouped into three modules, SOFT1, SOFT2, and SOFT3.

Module SOFT1 simulates the four basic single precision floating-point instructions, viz. add, subtract, multiply and divide. The corresponding entries to the module are FADDF, FSUBF, FMLTF, and FDIVF.

In module SOFT2, we simulate the instructions for fixed integer, fixed fraction, and negate with entries FFIXI, FFIXF, and FNEGF respectively.

Module SOFT3 simulates the instructions float integer, float fraction, and standardise. The corresponding entries into the module are FFLTI, FFLTF, and FSTND (or STEXP). There are two entries for the standardise operation. The entry 'FSTND' is the entry for standardising unnormalised floating-point numbers. 'STEXP' is the entry for standardisation if mantissa is given as a 32-bit extended mantissa (in register E) and exponent in register Y.

The results are returned as standardised single precision floating-numbers and are stored in register AB (or E). If overflow, underflow, no fixed or zero divide condition occurs the value of the first operand is returned.

NOTATION IN FLOWCHARTS:

We denote (AB), (Y) to represent the content of register AB and register Y respectively.

The flowcharts for addition/subtraction, multiplication, division and standardisation are given in figures 8, 9, 10, 11 in Chapter One respectively. The flowcharts for the rest of the procedures are given in figure 1a, 1b, and 1c in Appendix 1.

```
MBLER        MODULE  FSOFT1

         ZADM
         ZELS
         ZSLS
         ZELM
         ZMOD FSOFT1
         SENT=FMLTF=2=2;
         SETA L0
         JUMP S(UNPK1-*)
         SENT=FADDF=2=2;
         SETA L2
         JUMP S(UNPK1-*)
         SENT=FSUBF=2=2;
         SETA L3
         JUMP S(UNPK1-*)
         SENT=FDIVF=2=2;
         SETA L1
UNPK1;                                  /SET UP DATA AREA
         SRFS=TEMP=2
         SPAR=OPND1
         SPAR=OPND2
         SWRK=9=2
         SLCL=REXPON=1
         SLCL=FLG=1
         SLCL=COUNT=1
         STAS P(FLG)
         SETA L2
         STAS P(COUNT)                  /SET UP COUNTER
         LDRA P                         /LOAD A WITH CONTENT OF P, ;
                                        /  SET UP BASE ADDRESS IN P0
         ADDA L(COUNT+1)
         STAS P0
         CLRA
         STAS P1                        /SET INDEX =0
         SETE P(OPND1)                  /OPERAND1 ..UNPACK
UNPK2    LDRY B
         EXRA Y
         ANDA L[FF]                     /EXTRACT EXPONENT
         STAS M0 M1                     /EXP1 IN M0+0,  EXP2 IN M0+3
         INCS P1
         SETA P(FLG)                    /SET  A=FLG
         SZBA L14                       /TEST BIT 14  IF =0, MLT OR DIV.
         JUMP S(ADSB-*)                 /NO ADJUSTMENT NEEDED
         LSRB L8
         LSLB L7                        /GET LSF OF OPERAND
         SOBY L0                        /TEST IF MANTISSA POSITIVE.
         JUMP S(STORE-*) ;              /POSITIVE,NO RE-ARRANGEMENT
         SFCB ZZ                        /TEST IF B= ZERO
         JUMP S(STORE-*)                /POSITIVE, SIGN O.K.
         RSOB L0                        /NO..PUT IN SIGN BIT FOR NEG.NUMBER
         ADMY L1
         JUMP S(STORE-*)
ADSB     LSRB L8
         LSLB L8                        /NO REARRANGEMENT FOR ADD,SUBT.
STORE    LDRA Y                         /PUT Y IN A
         STAS M0 M1                     /STORE
         INCS P1                        /INDEX REGISTER INCREASE BY ONE
```

```
MBLER          MODULE  FSOFT1

               STBS M0 M1                    /STORE IN M0+2, M0+5
               INCS P1                       /INCREASE INDEX
               DECS P(COUNT)                 /DECREASE COUNTER
               JUMP S1                       /NOT ZERO  TAKE OPERAND2
               JUMP S(TFG-*) ;               /FINISH UNPACK
               SETE P(OPND2)
               JUMP S(UNPK2-*)
  TFG          SETA P(FLG)                   /TEST FLG
               LDRB A
               STCB IZ
               SETA SR                       /SWITCH
               JUMP ZA
               (FMLTF1)
               (FDIVF1)
               (FADDF1)
               (FSUBF1)
  FMLTF1;                                    /MULTIPLY STARTS
               SETA M0 +0
               ADDA M0 +3
               SUBA L128                     /EXP1+EXP2 -128(EXCESS)
               LDRY A                        /PUT REXPON INY
               SETA M0 +1                    /MSF(1)
               MLTA M0 +5                    /MSF(1)*LSF(2)
               ASRE L14
               STES P(TEMP)                  /STORE IN TEMP
               SETA M0 +4                    /MSF(2)
               MLTA M0 +2                    /MSF(2)*LSF(1)
               ASRE L14
               ADDE P(TEMP)
               STES P(TEMP)
               SETA M0 +1
               MLTA M0 +4                    /MSF(1)*MSF(2)
               SZBA L0                       /TEST IF BIT 0  OF A=0
               JUMP S4                       /NO
               SOBA L1                       /TEST IF LARGEST FRACTION
               JUMP S2
               ADMY L1
               JUMP S1                       /INCREASE EXPONENT
               LSLE L1
               ADDE P(TEMP)
               SNAO
               JUMP S(SCALE-*)
               JUMP S(NORM-*)
  FADDF1;                                    /ADDITION..START
               BSZY L14                      /SET BIT 14 OF Y=0 TO INDICATE ADD
               JUMP S1
  FSUBF1;                                    /SUBTRACT..START
               BSOY L14                      /SET BIT 14 OF Y=1 IF SUBTRACT
               SETA M0 +0
               STAS P(REXPON)                /STORE  REXPON=EXP1
               BSZY L15
               SUBA M0 +3                    /EXP1-EXP2..=DIFF
               STCA PZ                       /TEST IF DIFF GE 0
               JUMP S1                       /NO
               JUMP S4                       /YES
               BSOY L15                      /SET BIT 15 OF Y =1
```

```
ABLER          MODULE  FSOFT1

               SETB M0 +3
               STBS P(REXPON)
               NEGA                          /REPLACE REXPON=EXP2. NEGATE  DIFF
               STAS P1                       /M..DIFF
               SUBA L23                      /TEST IF DIFF GE 23
               STCA PZ                       /SKIP IF GE 23
               JUMP S(LT23-*)
               SZBY L15                      /YES..TEST IF EXP1 GE EXP2
               JUMP S3
               SETE L0                       /CLEAR OPND1 IF EXP1 LT EXP2
               STES M0 +4                    /STORE OPND1
               JUMP S(FLAG-*)
               SETE L0
               STES M0 +1                    /CLEAROPND1=0 IF EXP1 LT EXP2
               JUMP S(FLAG-*)
LT23           SETA P1                       /PUT DIFF IN A
               SUBA L15
               STCA PP                       /SKIP IF GT 0
               JUMP S4
               STAS P(COUNT)
               SETB L15                      /STORE DIFF -15 IN P4  ..SET B=15
               STBS P1                       /STORE IN P1
               JUMP S2
               CLRA
               STAS P(COUNT)                 /CLEAR CONTENT IN COUNT
               SZBY L15                      /SKIP IF EXP1 GE EXP2
               JUMP S(VGTU-*)
               SETE M0 +4                    /SET MANTISSA OF OPND2 AND SHIFT
               ASRE M1
               ASRE M4
               STES M0 +4                    /SHIFT OPND2
               JUMP S(FLAG-*)
VGTU           SETE M0 +1
               ASRE M1
               ASRE M4
               STES M0 +1
FLAG           SNAO                          /CLEAR AO
               CLRA
               SETE M0 +1
               SOBY L14                      /TEST FLAG IF 1  SUBTRACT
               JUMP S2
               SUBE M0 +4                    /SUBTRACT
               JUMP S1
               ADDE M0 +4                    /ADD
               LDMY M2
               SNAO                          /TEST FOR MANTISSA OVER FLOW
               JUMP S(SCALE-*)
               JUMP S(NORM-*)
FDIVF1                                       /DIVISION..START
               SETA M0 +0
               SUBA M0 +3
               ADDA L129                     /EXP1-EXP2+128EXCESS
               LDRY A
               SETA P(OPND2)
               SFCA Z7                       /TEST IF ZERO DIVIDE
               JUMP S(CODE3-*)               / ERROR CONDITION RETURN OPND1
```

```
MBLER          MODULE  FSOFT1

          SETE P(OPND1)
          ASRE L8
          LSLE L7                    /OBTAIN MANTISSA OF OPERAND 1;
                                     / SHIFTED ONE PLACE RIGHT
          DIVE M0 +4                 /DIVIDED BY MSF(2)
          STAS P(TEMP)               /REMAINDER IN B;;
                                     /QUOTIENT IN TEMP

          CLRA
          EXRA B                     /PUT REMAINDER IN A
          STBS P(TEMP+1)             /CLEAR  CONTENT IN TEMP+1
          ASRE L1
          DIVE M0 +4                 /DIVIDE THE REMAINDER BY MSF(2)
          CLRB                       /CLEAR REMAINDER AND ADD TO TEMP
          ASRE L14                   /SHIFT RIGHT 14 PLACES
          ADDE P(TEMP) ;             /ADD TO TEMP
          STES P(TEMP)               /STORE BACK IN TEMP
          CLRB
          SETA M0 +5                 /LSF(2)
          ASRE L1
          DIVE M0 +4                 /LSF(2)DIVIDED BY MSF(2);
                                     /....CORRECTION TERM
          MLTA P(TEMP)               /TIMES
          ASRE L13
          STES M0 +4                 /STORE CORRECTION TERM IN  M0+4
          SETE P(TEMP)
          SUBE M0 +4                 /SUBTRACT CORRECTION TERM
          SNAO
          JUMP S(SCALE-*)
          JUMP S(NORM-*)
CODE3;                               /ZERO DIVIDE
          SETE P(OPND1)              /SET E= OPERAND1
          LDMY L3                    /SET Y=3 IF ZERO DIVIDE
          JUMP S0 I
          (FE1)
SCALE     ASRE L1
          ADMY L1
          SZBA L0
          JUMP S2
          BSOA L0                    /NEGATIVE
          JUMP S1
          BSZA L0                    /POSITIVE
NORM      STES P(TEMP)
          STCK =E
          SCFP=STEXP=Y
          SXIT=N=TEMP
          ZEND
:    NO WARNINGS:  ALP2
```

```
BLER        MODULE  FSOFT2

            ZELS
            ZSLS
            ZELM
            ZMOD FSOFT2
            SENT=FFIXI=1=2
            BSZY L1                     /SET BIT1 OF Y =0
            SETA L159                   /INTEGER 31 INTEGRAL PLACES
            JUMP S(STEXP1-*)
            SENT=FFIXF=1=2
            SETA L128                   /FRACTION..0 INTEGRAL PLACES
            BSOY L1                     /SET BIT 1 OF Y=1;
                                        /..USE AS MARKER FOR FFIXF
            JUMP S(STEXP1-*)
            SENT=FNEGF=1=2
            MRKA
STEXP1;                                 /SET UP DATA AREA
            SRES=RESULT=2
            SPAR=OPND
            SWRK=1=0
            SLCL=REXPON=1
            SFCA DD
            JUMP S(BRNEG-*)
            STAS P(REXPON)
            SETE P(OPND)
            ASRE L8
            LSLE L8                     /GET MANTISSA ONLY
            STES P(RESULT)              /STORE MANTISSA IN RESULT
            STCA NN                     /SKIP IF A LT 0
            JUMP S(BEGIN1-*)            /POSITIVE
            STCB ZZ                     /SKIP IF B=0
            JUMP S(BEGIN-*)             /NO..NOT ZERO
            BSOB L0                     /SET B=[8000]
            SERA B                      /COMPARE WITH A
            JUMP S(BEGIN-*)             /ANOT EQUAL TO [8000]
            ASRA L1                     /DE-STANDARDISE
            STAS P(RESULT)              /STORE IN RESULT
            SETA P(OPND+1)              /EXTRACT EXPONENT
            ANDA L[FF]
            ADDA L1                     /INCREASE EXPONENT
            JUMP S(BEGIN2-*)
BEGIN1;                                 /TEST FOR ZERO MANTISSA
            STCA ZZ                     /SKIP IF A=0
            JUMP S(BEGIN-*)
            SNRA B                      /SKIP IF NOT EQUAL
            JUMP S(LEAVE-*)             /A=B=0
BEGIN       SETA P(OPND+1)
            ANDA L[FF]
BEGIN2      LDRB A                      /PUT EXPONENT IN B
            SUBA P(REXPON)              /SUBTRACT..EXPONENT-159 OR(128)
            STCA PP                     /SKIP IF REXPON-159(128) .GT.0
            JUMP S(OK1-*)
            LDMY L0                     /ERROR CONDITION..NO FIX, SET Y=0
            SETE P(OPND)
            JUMP S0 I
            (FE1)                       /JUMP OUT OF ROUTINE
OK1         LDRA B
```

```
BLER        MODULE  FSOFT2

            SZBY L1                         /TEST IF FFIXI ORFFIXF
            JUMP S2                         /NO FFIXF
            SUBA L128                       /YES..FFIXI
            JUMP S1
            SUBA L97                        /FFIXF..EXPONENT -128-31
            STCA NZ                         /SKIP IF LE 0
            JUMP S(TNEG-*)
FL          CLRA
            CLRB                            /(AB)=0
            JUMP S(LEAVE-*)
TNEG        LDRA B
            SUBA P(REXPON)                  /EXPONENT -128 OR(159)
            LDRY A                          /(Y)=(A)
            SETE P(RESULT)                  /LOAD MANTISSA INTO E
AGAIN       SFCY ZZ                         /SKIP IF NOT ZERO
            JUMP S(LEAVE-*)                 /(AB)=EXTENDED MANTISSA
            ASRE L1                         /NOT ZERO ARITH.SHIFT 1 PLACE
            ADMY L1                         /INCREASE EXPONENT
            JUMP S(AGAIN-*)
BRNEG:                                      /FNEGF...START
            SNAO
            CLRA                            /CLEAR AO
            SETA P(OPND+1)
            ANDA L[FF]                      /EXPONENT
            STAS P(REXPON)
            SETE P(OPND)
            ASRE L8
            LSLE L8                         /GET MANTISSA
            NEGE
            SNAO                            /SKIP IF NO OVERFLOW
            JUMP S1                         /OVERFLOW
            JUMP S(FIN-*)                   /NO OVERFLOW
            LSRE L1                         /SHIFT RIGHT 1 PLACE
            INCS P(REXPON)                  /INCREASE EXPONENT
            STES P(RESULT)
            SETA L255                       /TEST FOR OVERFLOW
            SUBA P(REXPON)
            STCA NN                         /SKIP IF GE 0
            JUMP S(FIN1-*)
            LDMY L1                         /OVERFLOW ..ERROR CONDITION SET Y=1
            SETE P(OPND)
            JUMP S0 I
            (FE1)                           /JUMP OUT OF ROUTINE
FIN1        SETE P(RESULT)
FIN         LDMY M0 ;
            //LOAD EXPONENT BEFORE TO STANDARDIZE ROUTINE ENTRY (STEXP)
            STCK =E
            SCFP=STEXP=Y
LEAVE       STES P(RESULT)                  /EXIT FROM MODULE
            SXIT=N=RESULT
            ZEND
   NO WARNINGS:  ALP2
```

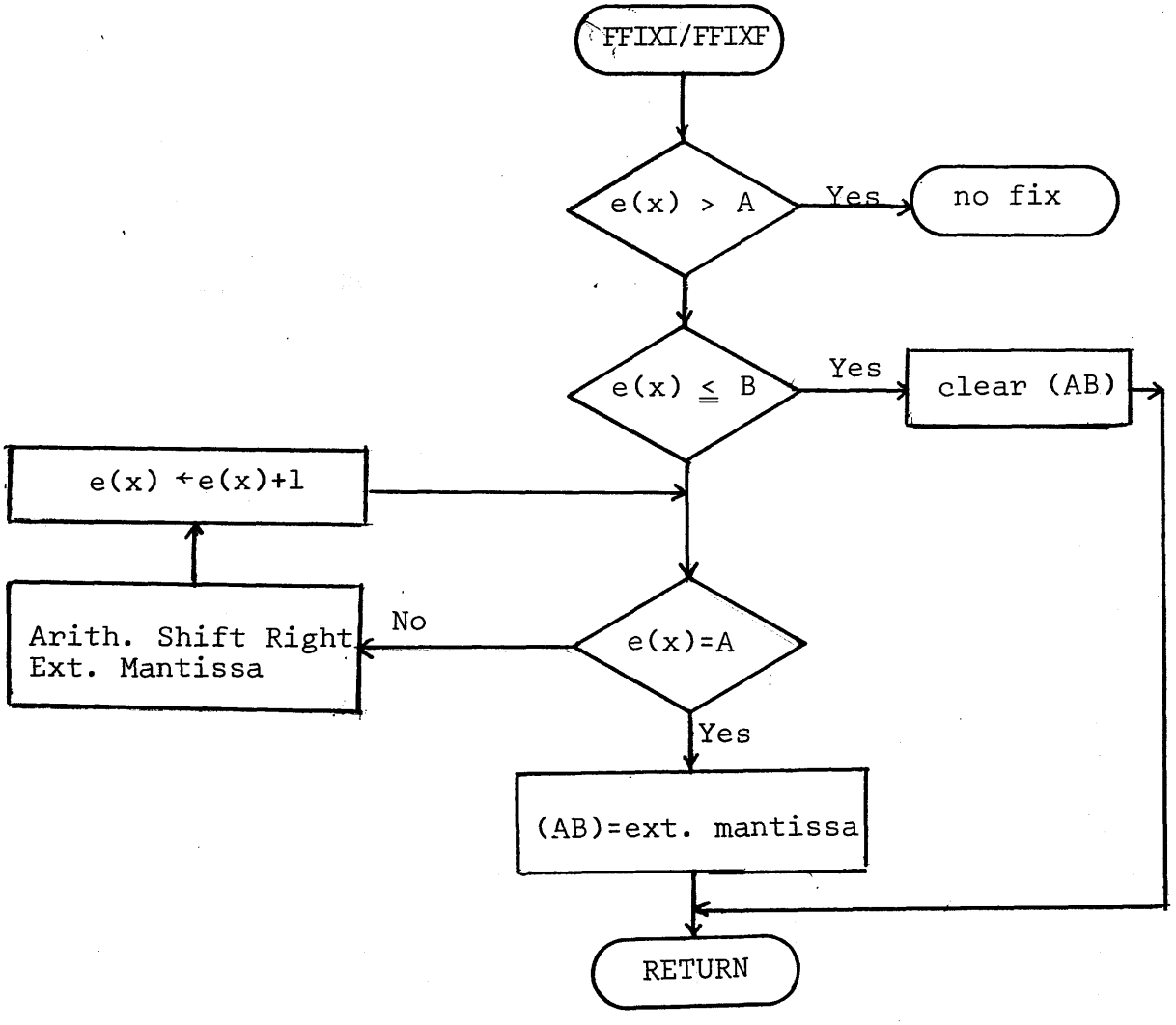
```
LER         MODULE  FSOFT3

            ZELS
            ZSLS
            ZELM
            ZMOD FSOFT3
            SENT=FFLTI=1=2
            CLRA
            LDRY A
            SETA L159                       /INTEGER HAVE 31 INTEGRAL PLACES
            JUMP S(STEXP2-*)
            SENT=FFLTF=1=2
            CLRA
            LDRY A
            SETA L128                       /FRACTION 0 INTEGRAL PLACES
            JUMP S(STEXP2-*)
            SENT=STEXP=1=2
            CLRA
            EXRA Y
            JUMP S(STEXP2-*)
            SENT=FSTND=1=2
            MRKA
            LDRY A
STEXP2                                      /SET UP MACROS FOR WORK SPACE
            SRES=RESULT=2
            SPAR=OPND
            SWRK=1=0
            SLCL=REXPON=1
            STCY NN
            JUMP S(STEXP3-*)
            SETB P(OPND+1)
            LDRA B
            SETH L0
            STAS P(OPND+1)
            LDRA B
            ANDA L[FF]                      /EXTRACTTHE EXPONENT
STEXP3      STAS P(REXPON)
            LDRY A
            SETE P(OPND)                    /FETCH THE UNSTANDARDISED NUMBER
            SERA B                          /TEST IF LOWER HALF= UPPER HALF
            JUMP S(NTZERO-*)                /NO,NOT ZERO IF DIFFERENT
            SFCA ZZ                         /TEST IF MANTISSA =0
            JUMP S(OUT-*)                   /RETURN 0 IN E
NTZERO:     //NUMBER NON-ZERO ..ENTER THE STANDARDIZE SEQUENCE
            STCA NN                         /TEST IF .NE.0
            JUMP S(POS-*)
FRNEG       SOBA L1                         /TEST IF BIT 1=BIT1 =1
            JUMP S(JOIN-*)
            LSLE L1                         /SHIFT UP ONE PLACE
            SBMY L1                         /DECREASE EXPONENT
            JUMP S(FRNEG-*)
POS:                                        /STANDARDISE ..A POSITIVE NUMBER
FRPOS       SZBA L1                         /TEST IF BIT 1=BIT0=0
            JUMP S(JOIN-*)                  /NO..STANDARDISED
            LSLE L1
            SBMY L1                         /DECREASE EXPONENT
            JUMP S(FRPOS-*)
JOIN:       //E NOW HOLDS A STANDARD FL.-PT. MANTISSA, NO EXPONENT:
```

```
.ER          MODULE  FSOFT3

         //TEST FOR OVERFLOW AND UNDER FLOW
         STES P(RESULT)
         STCY NN                          / TEST FOR UNDERFLOW
         JUMP S(OK2-*)
         LDMY L2                          /UNDERFLOW...SET Y=2
         SETE P(OPND)
         JUMP S0 I
         (FE1)
OK2      LDRA Y                           /TEST FOR OVERFLOW
         SUBA L[FF]
         STCA PP
         JUMP S(OK3-*)
         LDMY L1                          /ERROR CONDITION.. OVERFLOW , SET Y=1
         SETE P(OPND)
         JUMP S0 I                        /JUMP OUT OF ROUTINE
         (FE1)
OK3      SETE P(RESULT)
         ASRE L8
         LSLE L8                          /PUT LAST 8 BITS =0
         ADRB Y                           /ADD IN THE EXPONENT
OUT:                                      /EXIT FROM MODULE
         STES P(RESULT)
         SXIT=N=RESULT
         ZEND
  NO WARNINGS:  ALP2
```

FLOWCHART: FIX INTEGER/FIX FRACTION
(FFIXI/FFIXF



(Figure 1a)

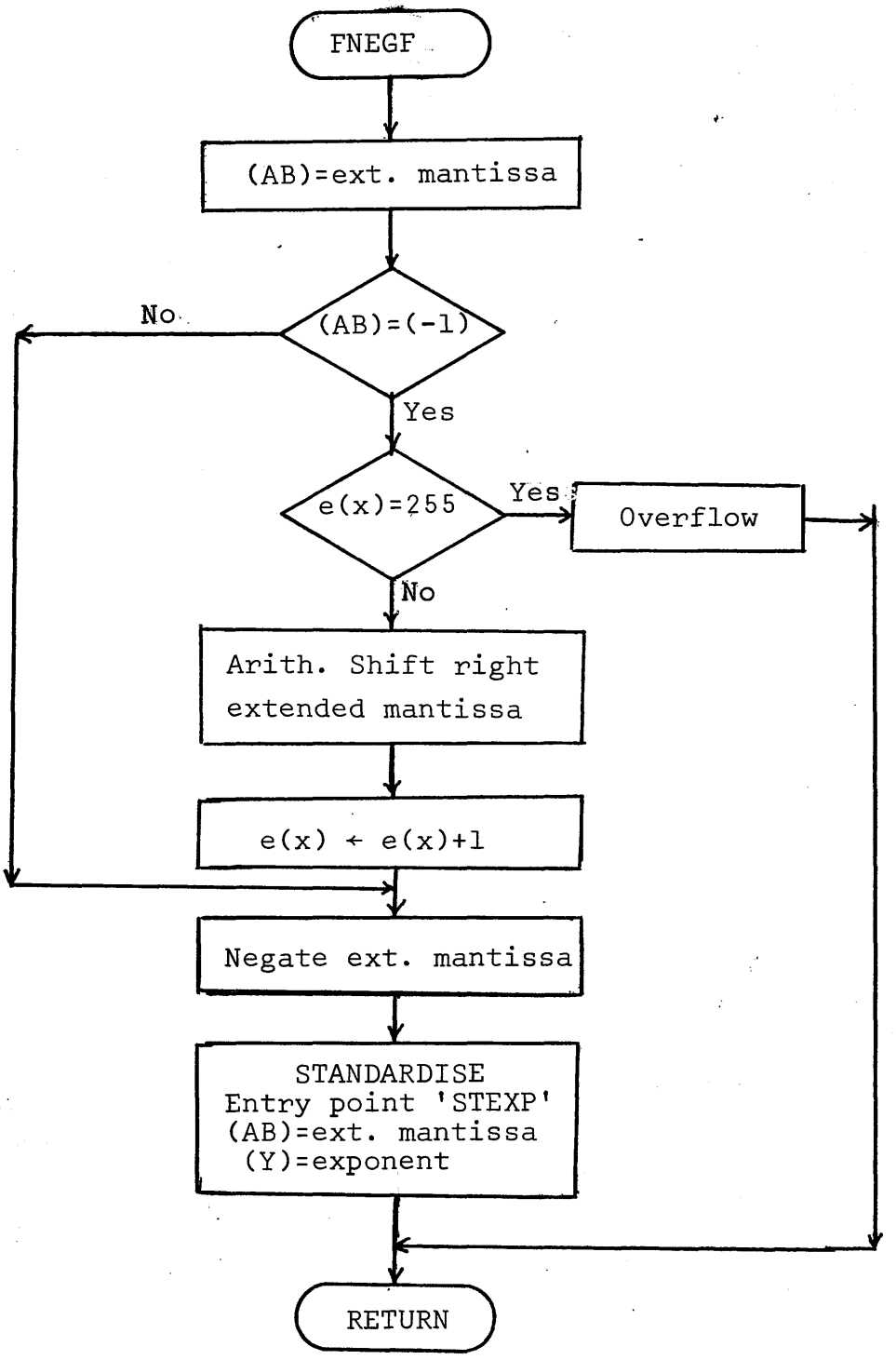
Note:
A = 12B + 31

B = 128

A = 128

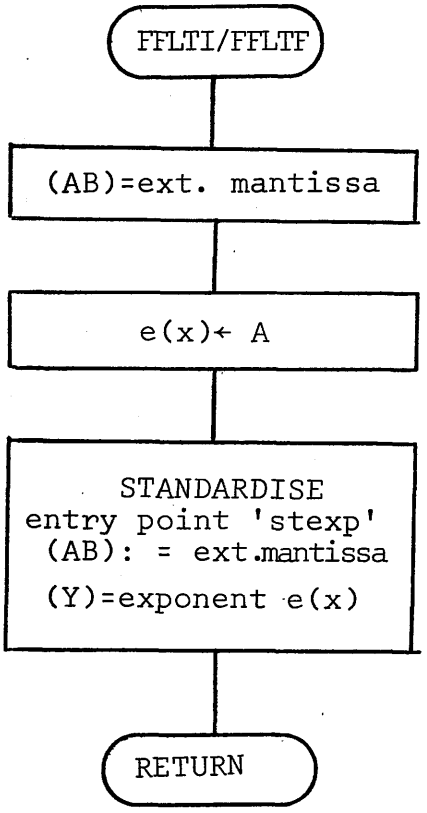
B = 128-31

If mantissa is (-1), it is de-standardised before a fixed instruction.


FNEGF
(AB)=ext. mantissa
(AB)=(-1)
No
Yes
e(x)=255
Yes
Overflow
No
Arith. Shift right
extended mantissa
e(x) ← e(x)+1
Negate ext. mantissa
STANDARDISE
Entry point 'STEXP'
(AB)=ext. mantissa
(Y)=exponent
RETURN

(Figure 1b)

FLOWCHART: FLOAT INTEGER/FLOAT FRACTION
(FFLTI/FFLTF)



(Figure 1c)

Note:
'Extended mantissa' is mantissa with extra 8 bits added to least significant end.

A = 128 + 31 for float integer routine

A = 128 for float fraction routine

APPENDIX 2

The ALGOL 60 procedures for double precision arithmetic and exact multiplication are taken from Dekker's paper (6).

The procedures work correctly if the single precision floating-point system is binary, single precision floating-point addition and subtraction are optimal, multiplication is faithful and no overflow or underflow occurs.

In the comments (x, xx), (y, yy) and (z, zz) denote nearly double precision numbers.

The algorithm for exact addition is

$$z = fl(x+y)$$
$$w = fl(x-z)$$
$$zz = fl(w+y)$$

In ALGOL 60 statements, these are written as:

```
z:=x+y
```

and

```
zz:=x-z+y
```

```
comment  add2 calculates the double precision sum of (x, xx)
         and (y, yy), the result being (z, zz);
procedure add2 (x, xx, y, yy, z, zz);
value     x, xx, y, yy;
real      x, xx, y, yy, z, zz;
begin real r, s;
        r:=x+y;
        s:=if abs(x)>abs(y) then
           x-r+y*yy*xx else y-r+x+xx+yy;
        z:=r+s;
       zz:=r-z+s
end       add2
comment   sub2 calculates the double precision of (x, xx)
          and (y, yy) the result being (z, zz);
procedure sub2(x, xx, y, yy, z, zz);
value     x, xx, y, yy;
real      x, xx, y, yy, z, zz;
```

```
begin real  r, s;
            r:=x-y;
            s:=if abs(x)>abs(y) then
            x-r-y-yy+xx else -y-r+x+xx-yy;
            z:=r + s;
            zz:=r-z+s
end         sub2;
comment     mult1 calculates the exact product of x and y,
            the result being the nearly double precision
            number (z, zz).  The constant should be chosen
            equal to 2 (t÷2)+1, where t is the number of
            binary digits in the mantissa;
procedure   mult1 (x, y, z, zz);
value       x, y;
real        x, y, z, zz;
begin real  hx, tx, hy, ty, p, q;
            p:=x constant;
            hx:=x-p+p; tx:=x-hx;
            p:=y×constant;
            hy:=y-p+p; ty:=y-hy;
            p:=hx×hy;
            q:=hx×ty+tx×hy;
            z:=p+q;
            zz:=p-z+q+tx×ty;
end         mult1 ;
comment     mult 2 calculates the double precision product
            of (x, xx) and (y, yy), the result being (z, zz);
procedure   mult 2(x, xx, y, yy, z, zz);
value       x, xx, y, yy; real x, xx, y, yy, z, zz;
begin real  c, cc;
            mult 2(x, y, c, cc);
            cc:=x×yy+xx×y+cc;
            z:=c+cc;
            zz:=c-z+cc;
end         mult 2
comment     div2 calculates the double precision quotient
            of (x, xx) and (y, yy), the result being (z,zz);
procedure   div2(x, xx, y, yy, z, zz);
value       x, xx, y, yy;
real        x, xx, y, yy, z, zz;
```

```
begin real  c, cc, u, uu;
            c:=x/y;
            mult2(c, y, u, uu);
            cc:=(x-u-uu+xx-c×yy)/y;
            z:=c+cc;
            zz:=c-z+cc
end         div2
```